Conception d'un noyau de vérification de preuves pour le $\lambda\Pi$-calcul modulo

Mathieu Boespflug

Thèse de doctorat
Spécialité INFORMATIQUE

Conception d'un noyau de vérification de preuves pour le λΠ-calcul modulo

Présentée et soutenue publiquement par

Mathieu Boespflug

le 18 janvier 2011, devant le jury composé de

Gilles	DOWEK	*Directeur de thèse*
Olivier	DANVY	*Rapporteurs*
Pierre-Étienne	MOREAU	
Ulrich	BERGER	*Examinateurs*
Benjamin	GRÉGOIRE	
François	POTTIER	
Benjamin	WERNER	

Thèse financée par

Composé avec ConTEXt version 2011.02.20 20 :34 tournant sur LuaTeX version 0.65.0.

« *Seek freedom and become captive of your desires. Seek discipline and find your liberty.* »
— Frank Herbert, *Dune*.

Remerciements

Au commencement, *ante* baccalauréat, il y avait une lettre grecque peu familière qui revenait à longueur de page d'un article de *Science et Vie* énonçant « la vraie nature de l'intelligence » (Poirier, 2002). Que de stupéfaction à l'idée que seules trois formes (dont la lettre λ) postulant l'existence d'adresses et la faculté de remplacer une adresse par un objet, suffisent à exprimer tant de calculs que Jean-Louis Krivine se mettait à imaginer que ces trois formes sont à la base du cerveau humain et de la conscience même. Presque 10 ans plus tard, *post* études supérieures, c'est finalement de ce même λ dont il s'agira pour beaucoup dans cette thèse, certes non pour expliquer l'Homme mais pour l'expliquer à la Machine.

Je tiens tout d'abord à remercier mon directeur de thèse Gilles Dowek, qui a su offrir le cadre scientifique nécessaire et veiller à ce que ma thèse ne diverge pas pour ne s'évaluer qu'en $\perp$, tout en m'accordant une très grande liberté scientifique. Je suis très reconnaissant pour sa disponibilité sans faille au cours de ces 3 années de thèse qui ont beaucoup bénéficié de son immense sagesse ainsi que de son sens de la synthèse et de la pédagogie.

Cette thèse serait toute autre sans l'apport de Olivier Danvy. Tel un Saunders Mac Lane pour qui les morphismes sont au moins aussi intéressants que les objets qu'ils connectent, Olivier m'a enseigné l'impératif de la simplicité et de l'esthétisme dans toutes mes recherches. C'est donc une grande joie pour moi de l'avoir parmi les rapporteurs de cette thèse. Olivier a apporté un oeil attentif et minutieux à cette thèse, ce qui m'a permis d'en améliorer de nombreux détails, combler quelques omissions et corriger quelques coquilles.

J'ai commencé ma thèse sous le signe de la réécriture. Aussi je suis heureux d'avoir également Pierre-Étienne Moreau comme rapporteur, dont le regard expert très syntaxique et opérationnel a soulevé de nombreuses questions et suscité quelques clarifications.

C'est un grand honneur pour moi que de compter Ulrich Berger, un des pères fondateurs de la normalisation par évaluation, parmi le jury. Parmi tous ces géants, je m'appuie tout particulièrement sur les épaules de Benjamin Grégoire dont la thèse a été le point de départ de la mienne. Merci à François Pottier pour l'intérêt qu'il a manifesté pour ce travail et ses nombreuses remarques sur le manuscrit, ainsi qu'à Benjamin Werner, chef d'équipe de mon berceau scientifique, l'équipe Typical.

J'ai ainsi grandi au sein d'une équipe de recherche chaleureuse dont je remercie tous les membres pour avoir contribué au fil de ces trois années à un environnement de travail scientifiquement riche et convivial. Je pense en particulier aux nombreuses discussions scientifiques avec Hugo Herbelin, Bruno Barras, Arnaud Spiwack et bien d'autres, au 1018 canal historique

(Denis Cousineau, Élie Soubiran, Bruno Bernardo), les débats de programmation à grands coups de tableau blanc avec Maxime Dénès, stagiaire à l'époque, sans oublier les trolls poilus de mi-journée avec grand renfort de l'équipe Sysres (Matthieu et James). Avec une mention toute particulière pour Jean-Marc Notin, qui n'a pas hésité à me prêter sa calculette sous Debian durant toute la durée de la rédaction du manuscrit, alors que la mienne succombait sous les coups de la loi de Murphy dès le jour 1. Enfin, je dois beaucoup aux demoiselles pleines de ressources pour parer à la tête en l'air que je suis, Lydie, Cindy, Valérie, Isabelle, Marie-Jeanne, Corinne, Évelyne et Catherine.

Merci à ma mère, sans la présence et les conseils de qui cette thèse n'aurait jamais commencé, et à mon père, qui m'a donné le goût de l'informatique.

Merci à Theodora, για την υποστήριξη και την αγάπη σου.

Abstract

In recent years, the emergence of feature rich and mature interactive proof assistants has enabled large formalization efforts of high-profile conjectures and results previously established only by pen and paper. A medley of incompatible and philosophically diverging logics are at the core of all these proof assistants. Cousineau et Dowek (2007) have proposed the $\lambda\Pi$-calculus modulo as a universal target framework for other front-end proof languages and environments. We explain in this thesis how this particularly simple formalism allows for a small, modular and efficient proof checker upon which the consistency of entire systems can be made to rely upon.

Proofs increasingly rely on computation both in the large, as exemplified by the proof of the four colour theorem by Gonthier (2007), and in the small following the SSReflect methodoly and supporting tools. Encoding proofs from other systems in the $\lambda\Pi$-calculus modulo bakes yet more computation into the proof terms. We show how to make the proof checking problem manageable by turning entire proof terms into functional programs and compiling them in one go using off-the-shelf compilers for standard programming languages. We use untyped normalization by evaluation (NbE) as an enabling technology and show how to optimize previous instances of it found in the literature.

Through a single change to the interpretation of proof terms, we arrive at a representation of proof terms using higher order abstract syntax (HOAS) allowing for a proof checking algorithm devoid of any explicit typing context for all Pure Type Systems (PTS). We observe that this novel algorithm is a generalization to dependent types of a type checking algorithm found in the HOL proof assistants enabling on-the-fly checking of proofs. We thus arrive at a purely functional system with no explicit state, where all proofs are checked by construction. We formally verify in Coq the correspondence of the type system on higher order terms lying behind this algorithm with respect to the standard typing rules for PTS. This line of work can be seen as connecting two historic strands of proof assistants : LCF and its descendents, where proofs of untyped or simply typed formulae are checked by construction, versus *Automath* and its descendents, where proofs of dependently typed terms are checked *a posteriori*.

The algorithms presented in this thesis are at the core of a new proof checker called Dedukti and in some cases have been transferred to the more mature platform that is Coq. In joint work with Denes, we show how to extend the untyped NbE algorithm to the syntax and reduction rules of the Calculus of Inductive Constructions (CIC). In joint work with Burel, we generalize previous work by Cousineau et Dowek (2007) on the embedding into the $\lambda\Pi$-calculus modulo of a large class of PTS to inductive types, pattern matching and fixpoint operators.

Résumé

Ces dernières années ont vu l'émergence d'assistants interactifs de preuves riches en fonctionnalités et d'une grande maturité d'implémentation, ce qui a permis l'essor des grosses formalisations de résultats papier et la résolution de conjectures célèbres. Mais autant d'assistants de preuves reposent sur presque autant de logiques comme fondements théoriques. Cousineau et Dowek (2007) proposent le $\lambda\Pi$-calcul modulo comme un cadre universel cible pour tous ces environnement de démonstration. Nous montrons dans cette thèse comment ce formalisme particulièrement simple admet une implémentation d'un vérificateur de taille modeste mais pour autant modulaire et efficace, à la correction de laquelle on peut réduire la cohérence de systèmes tout entiers.

Un nombre croissant de preuves dépendent de calculs intensifs comme dans la preuve du théorème des quatre couleurs de Gonthier (2007). Les méthodologies telles que SSReflect et les outils attenants privilégient les preuves contenant de nombreux petits calculs plutôt que les preuves purement déductives. L'encodage de preuves provenant d'autres systèmes dans le $\lambda\Pi$-calcul modulo introduit d'autres calculs encore. Nous montrons comment gérer la taille de ces calculs en interprétant les preuves tout entières comme des programmes fonctionnels, que l'on peut compiler vers du code machine à l'aide de compilateurs standards et clé-en-main. Nous employons pour cela une variante non typée de la normalisation par évaluation (NbE), et montrons comment optimiser de précédentes formulation de celle-ci.

Au travers d'une seule petite modification à l'interprétation des termes de preuves, nous arrivons aussi à une représentation des preuves en syntaxe abstraite d'ordre supérieur (HOAS), qui admet naturellement un algorithme de typage sans aucun contexte de typage explicite. Nous généralisons cet algorithme à tous les systèmes de types purs (PTS). Nous observons que cet algorithme est une extension à un cadre avec types dépendants de l'algorithme de typage des assistants de preuves de la famille HOL. Cette observation nous amène à développer une architecture à la LCF pour une large classe de PTS, c'est à dire une architecture où tous les termes de preuves sont corrects par construction, *a priori* donc, et n'ont ainsi pas besoin d'être vérifié *a posteriori*. Nous prouvons formellement en Coq un théorème de correspondance entre les système de types sans contexte et leur pendant standard avec contexte explicite. Ces travaux jettent un pont entre deux lignées historiques d'assistants de preuves : la lignée issue de LCF, à qui nous empruntons l'architecture du noyau, et celle issue de Automath, dont nous héritons la notion de types dépendants.

Les algorithmes présentés dans cette thèse sont au coeur d'un nouveau vérificateur de preuves appelé Dedukti et ont aussi été transférés vers un système plus mature : Coq. En collaboration avec Dénès, nous montrons comment étendre la NbE non typée pour gérer la syntaxe et les règles de réduction du calcul des constructions inductives (CIC). En collaboration avec Burel, nous généralisons des travaux précédents de Cousineau et Dowek (2007) sur l'encodage dans le $\lambda\Pi$-calcul modulo d'une large classe de PTS à des PTS avec types inductifs, motifs de filtrage et opérateurs de point fixe.

Table des matières

Introduction

La preuve est une chose trop importante pour la laisser aux mathématiciens. À ces derniers la démonstration : le discours et le raisonnement dont la finalité est la construction d'une preuve. La démonstration est un processus. La preuve est un objet. Comme tout autre objet, une preuve n'a de sens que si l'on décrète au préalable les conditions de son existence et sa forme. De fait, un objet physique que l'on peut toucher n'existe que si celui-ci est un ensemble de briques élémentaires agencées en accord avec les lois fondamentales de l'univers. Cet agencement nous donne la forme de l'objet. De même, une preuve est une suite de symboles encodant un ensemble d'hypothèses de départ, une assertion, et une liste d'applications des règles élémentaires de la logique établissant que l'assertion est une conséquence des hypothèses données. Mais pourquoi cette forme particulière de preuves ? Et quelles sont les règles élémentaires de la logique ? Après tout, existe-t-il qu'une seule logique mathématique ? Cette thèse fera mention par exemple de trois notions de preuves et d'une multitude de logiques proposées par logiciens et informaticiens au fil du XX^{e} siècle. L'objectif de cette thèse est de permettre l'interopérabilité des démonstrations par traduction des preuves d'une forme vers une autre, d'une logique vers une autre, d'une manière entièrement mécanique et vérifiable par ordinateur. Le mathématicien s'en trouve délesté de la charge lourde de vérifier que les démonstrations de ses pairs sont justes : il lui suffira de demander la réification de ces démonstrations en des preuves dont la forme et la logique sera du choix de ses pairs. À l'ordinateur de vérifier ces preuves.

La prise de risques mal maîtrisée est une contingence de la complexité croissante de la technologie dans notre vie quotidienne. Les risques de dysfonctionnements systémiques augmentent au fur et à mesure que l'interdépendance des composantes d'un système augmente. Le 21 avril 2010 vers 14 :00 UTC, plusieurs millions d'ordinateurs tournant sous une version particulière d'un système d'exploitation grand public perdaient leur accès à Internet. Plusieurs hôpitaux dans le Rhode Island renvoyaient des patients des salles d'urgence alors qu'une grande chaîne de supermarchés australienne fermait ses magasins dans les parties sud et nord du pays car les caisses automatiques tombaient en panne simultanément. Un seul bug informatique en était la cause : une mise à jour d'un logiciel antivirus provoquant la suppression accidentelle d'un fichier essentiel (McCullagh, 2010).

Cet incident met bien sûr en évidence la nécessité de la mise en place de protocoles de tests stricts et des mesures palliatives en cas de problème, sur le modèle des mécanismes de secours

en cas de pannes mécaniques dans les industries aéronautiques et automobiles. Tout système physique est soumis à des aléas imprévisibles dont on ne peut qu'espérer limiter l'impact. Mais les logiciels embarqués dans de tels appareils sont, eux, des objets déterministes dont on peut bien souvent modéliser le comportement (la sémantique) de façon arbitrairement précise. Par conséquent, il est envisageable d'élever au rang de vérité mathématique toute propriété de ces logiciels qui aura été démontrée sur les modèles de leur comportement – en particulier l'absence de bugs.

Cependant, les modèles de logiciels sont des objets mathématiques difficiles à manipuler et souvent de très grande taille. Les cahiers des charges le sont souvent aussi. Raisonner sur ces objets demande bien vite une impressionnante virtuosité technique. Ainsi la validité des démonstrations faites à la main de propriétés que l'on souhaite établir devient hasardeuse et sa vérification ardue. Il est intéressant alors de mettre les ordinateurs à contribution. Si le travail de démonstration n'est qu'en partie automatisable, la vérification de la preuve qui en résulte peut, quant à elle, être un processus entièrement mécanique.

Suivant la découverte par Boole (1854) que la notion de vérité peut être un objet mathématique à part entière, que l'on peut manipuler selon les lois de l'algèbre, le XIX^{e} siècle voit l'émergence de philosophies formalistes des mathématiques qui cherchent à codifier les étapes élémentaires de raisonnement nécessaires pour exprimer en détail, et de manière infaillible, toutes les démonstrations des mathématiques d'alors. Les démonstrations deviennent des objets mathématiques, c'est-à-dire des preuves, dont l'étude engendre une nouvelle branche de la logique : la théorie de la démonstration, fondée par Frege (1879). Hilbert formalise ce programme de recherche en 1920 à la suite de la découverte de paradoxes dans quelques-uns des premiers systèmes, qui verra de nombreuses contributions de la part de Whitehead et Russell (1910), Peano (1889), Gentzen (1934) et Gödel (1931), notamment. Avec cette formalisation des règles du jeu de la démonstration, les jalons sont déjà posés pour permettre, plus tard, la vérification de preuves où toutes les étapes logiques de la démonstration sont explicites par des ordinateurs, selon des règles simples, précises et bien étudiées.

Avant même la mise sous tension des premiers ordinateurs, Post (1936), Turing (1936) et Church (1936) formalisent simultanément et indépendamment la notion de calcul. Church (1940) remarque qu'une restriction de son modèle de calcul, le λ-calcul simplement typé, allié à un système de déduction à la Hilbert (1927) peut servir de cadre particulièrement simple et expressif pour écrire des propositions mathématiques et les démontrer, sans souffrir de paradoxes identifiés au tournant du siècle par Burali-Forti et Russell. L'approche employée par Church consiste à classifier les programmes par leur type, par exemple

$$\mathsf{plus} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

pour le programme qui calcule l'entier naturel qui correspond à l'addition des deux entiers naturels donnés en arguments. Les programmes de Church peuvent modéliser des fonctions mathématiques. Cette classification permet d'interdire l'écriture de formules insensées telles que l'application de la fonction plus à des objets mathématiques qui ne sont pas des entiers naturels. Curry et Feys (1958), Howard (1980) et De Bruijn (1970) rapprochent plus encore ces deux notions *a priori* distinctes que sont le calcul et la déduction, en remarquant que l'on peut lire les types des fonctions comme des propositions logiques. De manière toute aussi surprenante, les suites d'étapes de déduction dans une preuve correspondent à des programmes. Cette dualité entre types et propositions d'une part, et preuves et programmes d'autre part, souvent appelée « isomorphisme de Curry-Howard », offre un cadre idéal pour raisonner sur les programmes que l'on retrouve dans nos moyens de transport et l'équipement médical. Puisque la spécification que doit respecter un programme est une proposition, on peut extraire d'une preuve de cette proposition un programme respectant cette spécification, gratuitement.

L'identification de cet isomorphisme appuie l'élaboration de systèmes formels de plus en plus expressifs, de manière à permettre la formulation de spécifications de plus en plus précises et complexes sous la forme de formules mathématiques (propositions). Ces formules-spécifications sont typiquement bien plus complexes que les formules usuellement rencontrées en mathématiques, où l'on travaille le plus souvent dans des logiques simples mais en supposant en contrepartie des axiomes puissants. À titre d'exemple, la théorie des ensembles de Zermelo-Fraenkel, qui sert de base théorique à presque toutes les mathématiques, est une théorie de la logique du premier ordre avec une seule sorte. Les années 70 et 80 voient ainsi fleurir de nombreux systèmes formels de plus en plus expressifs, tous basés sur la théorie des types de Church (elle-même basée sur la théorie des types ramifiés de Russell), à l'image du Système F de Girard (1971) (découvert indépendemment par Reynolds (1974)), de la théorie de types de Martin-Löf (1984), le Calcul des Constructions de Coquand et Huet (1988) et son extension aux types inductifs (Coquand et Paulin, 1990) et à une hiérarchie d'univers (Luo, 1989).

Cette période marque aussi l'apparition des premiers assistants de preuve, comme LCF (Gordon et al., 1979) et Automath (De Bruijn, 1970). Une multitude d'autres suivent. Certains ne sont aucunement basés sur la théorie des types. Parmi ceux qui le sont, tous ne souscrivent pas à l'isomorphisme de Curry-Howard, dont la pertinence peut être fonction des domaines de spécialité (mathématiques pures, preuves de programmes, vérification de circuits électroniques, etc.) des assistants de preuve. De ce choix découle en partie d'autres choix de conception de ces systèmes, en premier lieu desquels le choix de la puissance expressive de la logique implémentée. De nombreux assistants (tels PVS, HOL, Isabelle ou Twelf) préfèrent offrir à l'utilisateur une logique moins expressive, tantôt plus facile à apprendre, tantôt offrant une métathéorie plus

simple ou avec certaines propriétés désirables, ou encore admettant une implémentation d'une plus grande concision.

C'est dans ce contexte d'environnements hétérogènes que s'inscrit cette thèse. Signe que nous avons aujourd'hui à disposition des assistants de preuve matures pour automatiser certaines parties des démonstrations et vérifier les preuves, nous assistons à l'émergence de projets de formalisations ambitieux comme celle du théorème des quatre couleurs (Gonthier, 2007), d'un compilateur complet (projet CompCert (Leroy, 2007)), de la résolution de la conjecture de Kepler (projet Flyspeck (Hales, 2005)) ou encore de grosses formalisations d'algèbre et de topologie (projet ForMath (Coquand, 2010)). L'envergure des ambitions implique une collaboration internationale entre équipes disparates dont l'expertise peut porter sur des assistants de preuves différents. En tout état de cause, il devient intéressant de pouvoir partager les banques de théorèmes développées dans chaque assistant avec un autre assistant.

Les problèmes d'hétérogénéité sont monnaie courante en informatique. Le raccord de n systèmes à chacun d'entre eux implique souvent d'adapter les impédances. Plutôt que de faire n^2 raccords, une solution plus économique consiste à raccorder tout le monde à un système, puis ce système à tous les autres systèmes. Il s'agit donc ici de concevoir un format commun de preuves, puis d'établir des traductions de tous les formats de preuves existants vers cet unique format. Un seul vérificateur de preuves est alors nécessaire pour vérifier le résultat de n assistants.

Ce format doit être en mesure d'accommoder toutes les démonstrations faites par tous les assistants de preuve, aussi bien actuels que futurs. Nous avons évoqué plus tôt deux axes selon lesquels nous pouvons réfléchir. Aux extrémités de chacun de ces axes, se trouvent les deux solutions suivantes :

1. inventer un système dont la logique est tellement puissante que tous les autres systèmes sont des cas particuliers de ce super-système ;
2. ou abandonner le rêve d'une telle tour de Babel et opter pour un formalisme très simple comme celui de la logique du premier ordre utilisé en théorie des ensembles, en s'en remettant à des axiomes forts pour étendre l'expressivité (c'est-à-dire l'ensemble des théorèmes prouvables) du système en fonction des besoins.

De Bruijn (1991) plaidait déjà il y a vingt ans pour des systèmes simples (« le cadre »), quitte à en payer le prix par un plus grand nombre d'axiomes dans les preuves (le « livre »), arguant qu'un cadre fort mène nécessairement à des schismes entre systèmes au fil du temps et qu'un cadre faible est plus facile à implémenter et donc plus digne de confiance. De Bruijn défend un cadre faible mais canonique face à une myriade de cadres forts que l'on peut difficilement hiérarchiser.

Dowek (2009) remarque cependant que l'introduction d'axiomes est dommageable pour de nombreuses propriétés métathéoriques d'un système. En théorie de la démonstration, ces propriétés sont des conséquences naturelles d'une propriété fondamentale d'un système de déduction : que toute formule démontrable admet une preuve sans coupures, c'est-à-dire sans détours par des lemmes dans la preuve. Par exemple, dans la déduction naturelle intuitionniste de Gentzen (1934), une conséquence de l'élimination des coupures est que si une formule de la forme $A \vee B$ est prouvable, alors soit A est prouvable ou B est prouvable. En présence d'axiomes, il se peut que $A \vee B$ soit un axiome et donc que l'on ne puisse se prononcer ni sur la prouvabilité de A, ni sur celle de B. De même, la présence d'axiomes ne permet plus de déduire la propriété du témoin ou le fait qu'une preuve sans coupure se termine toujours par une règle d'introduction d'un connecteur logique à partir de la seule propriété d'élimination des coupures.

Il se trouve qu'en présence des seuls axiomes définissant l'arithmétique, la propriété de la disjonction est vraie pour les propositions closes, mais il faut pour cela étendre la notion de coupure. Pour d'autres ensembles d'axiomes, la propriété de la disjonction pourra aussi être vraie, mais l'établir demande une notion encore différente de coupure. En général, chaque nouvel axiome ajouté au système demande une nouvelle notion de coupure. Et chaque nouvelle notion de coupure demande une nouvelle preuve d'une nouvelle propriété d'élimination des coupures.

Afin d'éviter d'introduire de nouveaux axiomes qui demanderaient une nouvelle extension de la notion de coupure, Dowek et al. (2003) proposent d'introduire un troisième axe de réflexion dans la conception de ce format de preuves universel. La déduction modulo est un système de déduction naturelle extensible par des règles de réécriture arbitraires sur les formules. Pour exprimer une démonstration faite dans un cadre plus puissant, nous avons maintenant la possibilité d'ajouter une nouvelle règle de réécriture pour émuler la puissance de ce cadre, plutôt que de recourir à l'ajout de nouveaux axiomes dans le livre. La notion d'élimination des coupures du cadre initial est préservée ; les propriétés métathéoriques attenantes le sont aussi.

L'ajout de règles de réécriture n'est pas une solution magique. L'élimination des coupures dans une preuve revient à mettre cette preuve dans une forme simplifiée, canonique, étape par étape. Via l'isomorphisme de Curry-Howard, les preuves sont des programmes. Cette simplification étape par étape correspond donc à un calcul de ces programmes. Vue à travers la lentille de cet isomorphisme, l'extension de la notion de coupure en présence de nouveaux axiomes correspond à donner un comportement calculatoire aux axiomes, alors qu'auparavant ceux-ci « bloquaient » l'élimination des coupures. L'ajout de règles de réécriture est un moyen générique de donner un comportement calculatoire à des axiomes. Ainsi, deux notions différentes

d'élimination des coupures dans un autre cadre ne sont que deux ensembles différents de règles de réécriture en déduction modulo.

La déduction modulo nous paraît ainsi être le cadre idéal pour exprimer les preuves issues de démonstrations dans d'autres systèmes. Il s'agit d'un cadre simple dans lequel il est pourtant possible d'exprimer les preuves issues des démonstrations de nombreux autres systèmes sans compromettre les bonnes propriétés du cadre. Cousineau et Dowek (2007) montrent qu'il est possible de traduire toutes les preuves de tous les PTS fonctionnels (une classe particulièrement large de théories) dans un format de preuve pour la déduction modulo particulièrement facile à manipuler pour un ordinateur : le $\lambda\Pi$-calcul modulo.

Cette facilité ne doit rien au hasard. Il s'agit en fait d'un format de programmes, extension minime du λ-calcul originel de Church tel que proposé en 1936. Toujours par le même isomorphisme, il n'en est pas moins un format de preuves. L'essentiel de cette thèse portera sur les techniques mises en œuvre pour vérifier les preuves-programme du $\lambda\Pi$-modulo, aboutissant à une implémentation complète et efficace d'un vérificateur de preuves appelé Dedukti. Nous généraliserons également la classe des preuves que l'on peut vérifier dans le cadre de la déduction modulo aux preuves employant des schémas d'induction, en particulier celles du Calcul des Constructions Inductives produites par Coq.

Il existe déjà des traducteurs automatiques de preuves dans un format existant vers un autre format existant, tel que de HOL et Nuprl vers LF (Schürmann et Stehr, 2006), HOL vers Nuprl (Howe, 1996 et Naumov et al.,) Isabelle (Obua et Skalberg, 2006) et Coq (Keller et Werner, 2010), ou de PVS vers Nuprl (Allen et al., 2003). Il existe également divers propositions récentes de formats de preuves simplifiés pour de nombreuses théories de types dépendants (Altenkirch et al., 2010 et Coquand et al., 2009) qui permettent un traitement uniforme de fonctionnalités haut niveau tels que les types inductifs, les types coinductifs et différentes formes de récurrence et d'analyse par cas. Le formalisme $\Pi\Sigma$ de Altenkirch et al. vu à travers la lentille de Curry-Howard n'est cependant pas une logique cohérente, de sorte qu'il est nécéssaire de vérifier un certain nombre de conditions de bord sur les termes à typer pour conserver la pertinence de cette lentille. Il serait concevable d'utiliser de tels formats comme des formats universels et extensibles, mais nous arguons que le $\lambda\Pi$-calcul modulo est un cadre plus simple : il n'introduit pas d'impuretés au delà du λ-calcul pur, celles-ci étant facilement simulables grâce à la notion très générale de règle de réécriture. Le $\lambda\Pi$-calcul modulo est aussi plus modulaire : le formalisme de base donne une logique cohérente. Si nécessaires, les types inductifs et autres fonctionnalités doivent être simulés avec l'ajout de règles de réécriture qui compromettent potentiellement la cohérence, mais la cohérence est conservée tant que les règles respectent des critères simples.

Nous nous efforcerons dans cette thèse de donner tout son sens à l'isomorphisme de Curry-Howard en traitant les preuves exactement comme des programmes. Un programme se compile vers du code machine. Il s'optimise pour de meilleures performances et il arrive souvent qu'il soit transformé dans des formes plus régulières pour en faciliter l'analyse. Nous compilerons donc les preuves vers du code machine (chapitre 3). Nous appliquerons des optimisations standards pour de meilleures performances (section 3.6 et section 4.7.3) et nous appliquerons des transformations elles aussi standards des langages de programmation pour transformer les preuves dans des formes plus régulières pour en faciliter l'analyse (section 1.6.1). Bien que les correspondances preuves/programmes et formules/types soient bien connues et étudiées depuis plusieurs décennies, nous pensons qu'il s'agit là d'un aspect novateur de cette thèse. Si l'idée de réutiliser tout ou partie de technologies de compilation existantes a été mise en œuvre pour la première fois dans la thèse de Grégoire (2003) et reprise depuis dans quelques autres systèmes (Aehlig et al., 2008), il s'agissait alors de compiler ponctuellement certaines parties des preuves pour accélérer le test de conversion que l'on retrouve dans toutes les théories des types dépendants. Nous traiterons ici des preuves dans leur globalité comme des programmes, en ne revenant que ponctuellement à la théorie de la démonstration.

Plan de la thèse

Cette thèse est composée de 5 chapitres, traitant tour à tour de théorie de la démonstration et de calcul. Les deux premiers introduisent les notions et définitions qui nous seront utiles dans les chapitres suivants.

Le chapitre 1 présente diverses notions de calcul, sa modélisation par le λ-calcul de Church et s'attarde sur les stratégies d'implémentation de ce dernier. Nous tentons en particulier d'expliquer les gains colossaux en performance que peut permettre la compilation des termes du λ-calcul vers du code machine.

Le chapitre 2 présente diverses théories des types, du λ-calcul simplement typé aux *Pure Type Systems* de Barendregt (1991), ainsi que leurs propriétés métathéoriques.

Le chapitre 3 présente la première (à notre connaissance) compilation du test de conversion des théories des types dépendants vers du code natif, en réutilisant entièrement des compilateurs existants pour des langages de programmation fonctionnels. Cette stratégie d'implémentation offre de très bonnes performances pour certaines preuves dont la vérification serait intraitable sans compilation. Nous partons d'une interprétation de Mogensen (1992) adaptée des termes du λ-calcul vers les termes du λ-calcul, à partir de laquelle nous dérivons un auto-réduiseur plus simple et plus efficace que l'auto-réduiseur proposé par

Mogensen. Nous identifions l'algorithme obtenu comme une variante de normalisation par évaluation non typée. Nous montrons ensuite comment améliorer de manière significative la performance de la normalisation des termes nécessaire au test de conversion à l'aide d'une optimisation standard des interpréteurs de langages fonctionnels et en externalisant le plus de travail possible au compilateur. Nous intégrons la traduction des règles de réécriture du préprocesseur Moca (Blanqui et al., 2007) pour généraliser l'algorithme de conversion par évaluation à une congruence générée par un système de règles de réécriture arbitraire. Enfin, avec le concours de Maxime Dénès dans le cadre de son stage de M2, que j'ai encadré, nous étendons aussi cet algorithme aux termes du Calcul des Constructions Inductives de Coq ; extension qui a servi de base pour une implémentation complète et mature du test de conversion de Coq par compilation vers du code natif.

Le chapitre 4 présente une version des Pure Type Systems où le contexte est absent de tous les jugements de typage. Nous développons d'abord la métathéorie du système obtenu pour le λ-calcul simplement typé, de manière entièrement formalisée en Coq. Nous observons que le système obtenu est la variante du λ-calcul simplement typée servant à représenter les formules de HOL dans une architecture à la LCF. Nous généralisons ensuite ce résultat à tous les PTS. Nous obtenons d'une part un système de types qui est une variante des PTS compatible avec une architecture à la LCF. Nous montrons d'autre part comment extraire directement de ce système de types un algorithme de vérification simple et efficace de preuves encodées en HOAS. Cet algorithme est au cœur de Dedukti (mais étendu avec des règles de réécriture).

Le chapitre 5 traite de la mise en œuvre des chapitres précédents dans un vérificateur complet compilant les preuves en $\lambda\Pi$-modulo vers des programmes fonctionnels en Haskell. Nous présentons également des travaux communs avec Guillaume Burel portant sur la traductions des PTS en $\lambda\Pi$-modulo au Calcul des Constructions.

Chapitre 1
Le λ-calcul et la machine

Nous traiterons dans ce chapitre de langages permettant d'exprimer un calcul de façon abstraite, ainsi que des méthodes permettant d'effectuer ce calcul, étant donné son expression abstraite, sur des machines usuelles. La logique dans laquelle nous nous plaçons pour énoncer les définitions et théorèmes ainsi que pour écrire les démonstrations est la théorie des types, plutôt que la théorie des ensembles plus conventionnelle dans les traités mathématiques. Nous supposons acquises les notions usuelles en théorie des ensembles de relation, de fonction, d'ensemble et d'entier naturel. Leur encodage en théorie des types n'est pas essentiel pour la compréhension des définitions formelles de ce chapitre ; ainsi nous remettons au chapitre 2 son exposition.

1.1 Calcul et langage

L'informaticien considère que la notion de langage sous-tend celle de toute notion de calcul, car les calculs véritablement intéressants sont ceux dont on peut observer le résultat étant donné une configuration de départ que l'on peut fixer arbitrairement. On pourrait considérer qu'une montre analogique bien réglée effectue chaque seconde un calcul différent. On peut donc considérer la montre comme une machine à calculer. Même pour une machine aussi simple que celle-ci, le langage joue un rôle : il faut bien pouvoir lire le cadran de la montre pour que celle-ci ait une quelconque utilité. Plus encore, la montre est mue par un état interne (la tension d'un ressort, par exemple) qui guide le résultat du calcul à chaque instant. Cet état n'est pas arbitraire mais appartient à l'ensemble de tous le états possibles. Une grammaire génératrice de cet ensemble peut servir à le caractériser. Si d'aventure l'état interne de la montre échappait de cet ensemble, la montre ne serait plus en mesure de fonctionner. Ou tout du moins, les calculs successifs qu'elle effectuerait alors seraient *a priori* imprévisibles et incompréhensibles.

Mais les calculs effectués par cette montre bien réglée ne sont pas très intéressants pour deux raisons. La première est qu'à chaque instant t il n'est pas possible de changer la configuration de départ de la montre, c'est-à-dire l'heure courante, sauf à désynchroniser l'horaire affiché du temps réel ou savoir inverser ou accélérer la flèche du temps. La deuxième est que tous les calculs de la montre se suivent et se ressemblent beaucoup. Pour une observation donnée de l'état de la montre, chaque aiguille du cadran aura fait une rotation dont le signe et l'amplitude restent constants d'une seconde à l'autre. Il n'est pas possible de changer la montre pour lui demander d'effectuer un calcul autre que ceux pour lesquels elle a été conçue. À la racine du

problème, on trouvera que le langage interne à la montre est bien trop pauvre pour paramétrer le calcul effectué de manière à en changer significativement la nature.

Notons que le langage interne de la montre analogique est de nature physique, mais les détails de ce langage importent peu. En lieu et place d'un ressort et un jeu de roues, nous pourrions installer un circuit électronique allié à un morceau de quartz finement taillé de manière à vibrer à une fréquence précise sous l'effet d'un potentiel électrique. Bien que le langage interne de la montre en serait alors changé, le calcul effectué et les résultats observés peuvent être mis en bijection avec ceux de la montre analogique. La montre digitale *simule* ainsi la montre analogique. Nous pouvons de surcroît nous affranchir du caractère physique des langages internes aux montres et passer à un langage de symboles qu'il est possible de retranscrire sur une feuille de papier. Les calculs successifs de la montre analogique peuvent tout autant qu'avec la montre digitale être simulés sur papier, ou mentalement, à l'aide de chiffres et autres symboles.

En théorie de la calculabilité, on considère le plus souvent des langages que l'on caractérise comme des ensembles de chaînes de symboles. Les symboles forment donc les éléments de base de tout langage dans ce cadre. À contrario, nous considérons dans la suite de ce chapitre plusieurs formalismes de langages. Tantôt des langages portant sur des arbres (que l'on qualifiera de *formels*), tantôt des langages de chaînes de codes machine qu'un processeur peut interpréter comme une suite d'instructions et d'octets. L'objet de ce chapitre est de rappeler comment simuler un calcul exprimé à l'aide d'un langage d'arbre, par un calcul équivalent exprimé comme une suite d'instructions et d'octets. Il est entendu que tous les calculs exprimé à l'aide de ces chaînes sont eux-mêmes des simulations de phénomènes physiques bien réels se produisant au coeur même de supercalculateurs et autres téléphones mobiles. Ainsi, passer d'un langage à un autre permet de simuler sur ordinateur un calcul formel.

1.2 Calcul par réécriture

La section précédente s'est bien gardée de définir précisément ce qu'est un calcul, puisque la forme que peut prendre celui-ci en mathématiques et plus encore dans la nature est diverse. Mais si l'on part du postulat que tous les calculs de la nature sont en principe simulables par d'autres calculs[1], choisissons un cadre très restreint de calculs plus amènes à l'analyse, sachant que ces calculs peuvent simuler bon nombre de calculs de la nature, si ce n'est tous.

[1] Si les calculs simulants sont opérés par la machine universelle de Turing (1936), ce postulat est la thèse de Church-Turing-Deutsch, ou version forte de la thèse de Church-Turing. Deutsch l'énonce ainsi :

« Every finitely realizable physical system can be perfectly simulated by a universal model computing machine operating by finite means. »

(Deutsch, 1985)

Les notations suivantes concernant les relations binaires nous aideront par la suite. Nous supposons dans toute cette section l'existence d'un type A quelconque, pendant en théorie des types d'un ensemble en théorie des ensembles. Chacun des habitants de A est appellé *terme*.

Définition 1.1 Soit $(\longrightarrow)$ une relation binaire sur A. Pour tout a,$b : A$, nous écrivons $a \longrightarrow b$ pour $(\longrightarrow)\ a\ b$. Nous noterons également $(\longrightarrow^{0,1})$ la clôture réflexive de $(\longrightarrow)$, $(\longrightarrow^{+})$ la clôture transitive, $(\longrightarrow^{*})$ la clôture réflexive et transitive, et $(\equiv_{\longrightarrow})$ la clôture réflexive, symétrique et transitive. On peut aussi construire la réciproque de $(\longrightarrow)$, que nous noterons $(\longleftarrow)$. L'union de $(\longrightarrow)$ et de sa réciproque est notée par $(\longleftrightarrow)$.

L'énoncé du *Entscheidungsproblem* par Hilbert en 1928 requit la notion « d'algorithme » ou de « méthode effective de calcul ». Hilbert se demandait s'il existait une procédure mécanique pour distinguer les vérités mathématiques de l'ivraie. Le calcul énoncé comme un algorithme est un processus « mécanique » ou *effectif* d'exécution d'instructions les unes à la suite des autres. C'est cette notion de calcul pas à pas qui amène à énoncer le cadre formel suivant pour le calcul.

Définition 1.2 (Réduction) Nous dirons que a se *réduit* ou se *réécrit* en b si $a \longrightarrow^{*} b$. Une séquence finie de réductions pour $(\longrightarrow)$ est une suite finie $(a_i)_{0 \leq i \leq n}$ telle que pour tout $0 \leq i \leq n-1$, $a_i \longrightarrow a_{i+1}$. Une séquence infinie de réductions pour $(\longrightarrow)$ est une suite infinie $(a_i)_{i \in \mathbb{N}}$ telle que pour tout $i \in \mathbb{N}$, $a_i \longrightarrow a_{i+1}$.

Définition 1.3 (Forme normale) Soit $a : A$ et $(\longrightarrow)$ une relation binaire sur A. a est en *forme normale* pour $(\longrightarrow)$ si il n'existe pas de $b : A$ tel que $a \longrightarrow b$. Pour tout $c : A$ tel que $c \longrightarrow^{+} a$, nous dirons que a est une forme normale de c.

Il est intéressant de savoir s'il est possible d'atteindre une forme normale à partir d'un élément quelconque a, et plus encore si toute les suites de réductions partant de a mènent vers une forme normale. On distingue donc deux formes de normalisation, la dernière étant plus forte que la première et souvent beaucoup plus difficile à établir.

Définition 1.4 (Normalisation faible) Pour tout $a : A$, a est *faiblement normalisant* (noté $\mathcal{WN}_{\rightarrow} a$) pour une relation binaire $(\longrightarrow)$ si il existe une forme normale de a. Une relation $(\longrightarrow)$ est faiblement normalisante si $\forall a : A, \mathcal{WN}_{\rightarrow} a$.

Définition 1.5 (Normalisation forte) Soit $(\longrightarrow)$ une relation binaire sur A et $a : A$. On définit le prédicat $\mathcal{SN}_{\rightarrow} a$ inductivement par :

- pour tout a, si a est en forme normale alors $\mathcal{SN}_{\rightarrow} a$;
- si pour tout $a' : A$ tel que $a \longrightarrow a'$ alors $\mathcal{SN}_{\rightarrow} a'$, alors $\mathcal{SN}_{\rightarrow} a$.

Cette définition inductive du prédicat $\mathcal{SN}_{\rightarrow}$ correspond en théorie des ensemble au plus petit ensemble vérifiant les conditions énoncées.

On dira d'un système de réécriture défini comme une relation binaire $(\longrightarrow)$ sur A qu'il est faiblement normalisant si pour tout $a : A$ est faiblement normalisant ; *mutatis mutandis* pour un système fortement normalisant.

Un calcul par réécriture est une suite de réductions potentiellement infinie, commençant par un élément $a_0 : A$. Dans le cas où cette suite est de taille finie, le résultat du calcul est une forme normale de a_0. Le calcul à l'aide d'un automate (déterministe ou non), cadre formel plus usuel en informatique, est une instance de la réécriture où l'ensemble A est l'ensemble de tous les états de l'automate.

1.3 Confluence

Une instance particulièrement intéressante de la réécriture sont les calculs formels, où l'on voit souvent les termes comme des programmes. Chaque étape de réduction produit un nouveau programme. Le résultat final d'un programme est sa forme normale, si elle existe. Dans ce cadre, la *confluence* est une propriété particulièrement intéressante.

En effet, si un système de réécriture $(\longrightarrow)$ ne normalise que faiblement, alors il peut exister des termes à partir desquelles certaines suites de réductions arrivent à une forme normale alors que d'autres suites n'y parviennent pas. Même si $(\longrightarrow)$ est fortement normalisant, certains termes peuvent avoir plusieurs formes normales auxquelles on parvient par des suites de réductions distinctes. Il n'est pas aisé d'écrire des programmes en présence de ce non-déterminisme de la forme normale éventuellement atteinte. Dans la mesure où l'intérêt d'un calcul est son résultat final, ce non-déterminisme peut aussi être problématique pour la performance du calcul car il peut être nécessaire d'essayer toutes les suites de réductions avant de trouver une forme normale, ou la bonne forme normale.

Définition 1.6 (Confluence) Soit $(\longrightarrow)$ une relation binaire sur A.

a. Elle est confluente si

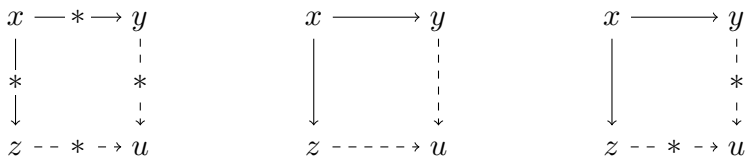
$$\forall x,y,z : A.\ x \longrightarrow^* y \wedge x \longrightarrow^* z \text{ implique } \exists u : A.\ y \longrightarrow^* u \wedge z \longrightarrow^* u.$$

b. Elle est fortement confluente si

$$\forall x,y,z : A.\ x \longrightarrow y \wedge x \longrightarrow z \text{ implique } \exists u : A.\ y \longrightarrow^{0,1} u \wedge z \longrightarrow^{0,1} u.$$

c. Elle est localement confluente si

$$\forall x,y,z : A.\ x \longrightarrow y \wedge x \longrightarrow z \text{ implique } \exists u : A.\ y \longrightarrow^* u \wedge z \longrightarrow^* u.$$



(a) (b) (c)

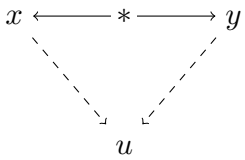
Remarquons que la confluence d'une relation correspond à la confluence forte de sa clôture réflexive et transitive.

Lemme 1.7 (Newman (1942)) *Une relation fortement confluente est toujours confluente.*

Dans le contexte du λ-calcul présenté ci-après, on retrouve souvent la propriété de Church-Rosser à propos de deux termes *convertibles*, c'est-à-dire deux termes reliés par la relation d'équivalence engendrée par une relation $(\longrightarrow)$.

Définition 1.8 (Church-Rosser (1936)) Une relation $(\longrightarrow)$ vérifie la relation de Church-Rosser si tous les termes convertibles ont un réduit commun :

$$\forall x,y : A.\ x \equiv_{\longrightarrow} y \text{ implique } \exists u : A.\ y \longrightarrow^* u \wedge x \longrightarrow^* u.$$



Lemme 1.9 *Une relation admet la propriété de Church-Rosser si et seulement si elle est confluente.*

Preuve. Voir (Kirchner et Kirchner, 1999 théorème 4.2). □

La propriété de Church-Rosser est fondamentale pour tester la convertibilité de deux termes efficacement ; nous y reviendrons à la section 3.1 du chapitre 3.

1.4 le λ-calcul

Le λ-calcul de Church (1936) est un système de réécriture et un modèle particulièrement simple de calcul. Dans sa forme pure, tout y est représenté comme une fonction, y compris les structures de données usuelles telles que les entiers naturels, les listes ou les arbres.

Définition 1.10 (λ-termes) L'ensemble des termes du λ-calcul est défini inductivement par la grammaire suivante :

$$\begin{array}{rcll} \mathsf{Var} & \ni & x, y, z & \\ \mathsf{Term} & \ni & M, N & ::= \quad x \mid \lambda x.\ M \mid M\ N \end{array}$$

où Var est un ensemble infini dénombrable de noms de variables. Un λ-terme est une variable, l'abstraction d'une variable dans un terme, ou l'application d'un terme à un autre. Par convention, l'application est associative à gauche et est prioritaire sur l'abstraction. Le terme $\lambda x.\ x\ y$ doit donc se comprendre comme $\lambda x.\ (x\ y)$ et non $(\lambda x.\ x)\ y$.

Le langage est spartiate mais étonnamment expressif — trop expressif même, puisque Kleene et Rosser (1936) démontrent que le λ-calcul considéré comme une logique est incohérent. Le problème vient du fait que le λ-calcul permet d'encoder toutes les fonctions effectivement calculables, alors même que certaines encodent quelques paradoxes, tel que le paradoxe de Russell[2]. De par sa puissance expressive et sa simplicité, le λ-calcul n'en est pas moins un formalisme idéal pour encoder nombre de fonctions mathématiques sans introduire d'axiomes supplémentaires.

Définition 1.11 En théorie des ensembles, pour tout $n \in \mathbb{N}$, un *entier de Church* est une fonction qui à toute fonction s associe la composition de s avec elle même itérée n fois :

$$\{n \in \mathbb{N} \mid \lambda z.\ \lambda s.\ \big(\bigcirc_{i=0}^{n} s\big)\ z\},$$

où $(\circ) = \lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x)$. $\ulcorner \cdot \urcorner$ est la fonction qui à un entier quelconque lui associe le λ-terme correspondant selon l'encodage de Church.
Dans une théorie des types avec une sorte $\mathbf{Type}$ imprédicative, un entier de Church est tout terme du type Church :

$$\mathsf{Church} \mathrel{\hat{=}} \forall A : \mathbf{Type}.\ A \to (A \to A) \to A$$

La définition en théorie des types est plus générale car les termes convertibles (voir la section 2.2) à un entier de Church sont aussi des entiers de Church.

Définition 1.12 Étant donné un encodage à la Church des entiers naturels, l'addition $(+)$ sur les entiers s'encode comme suit :

$$(+) \mathrel{\hat{=}} \lambda x.\ \lambda y.\ \lambda z.\ \lambda s.\ x\ (y\ z\ s)\ s$$

[2] Nous verrons comment résoudre ce problème dans le chapitre 2.

Il est pratique d'introduire une relation d'équivalence entre l'application d'une fonction à un argument et son résultat. Il est souhaitable par exemple d'identifier $\ulcorner 1 \urcorner + \ulcorner 1 \urcorner$ et $\ulcorner 2 \urcorner$. Church introduit en ce sens plusieurs règles de réduction. On obtient une relation d'équivalence en prenant la clôture réflexive, symétrique et transitive de ces relations de réduction.

Définition 1.13 (Variables libres, terme clos) *L'ensemble des variables libres* d'un terme est donné inductivement par les équations suivantes :

$$\begin{aligned} \mathcal{FV}(x) &= \{x\} \\ \mathcal{FV}(\lambda x.\ M) &= \mathcal{FV}(M) \setminus \{x\} \\ \mathcal{FV}(M\ N) &= \mathcal{FV}(M) \cup \mathcal{FV}(N) \end{aligned}$$

Un terme M tel que $\mathcal{FV}(M)$ est dit *clos* si $FV\ (M) = \emptyset$.

Définition 1.14 (Substitution) La substitution d'un terme N pour une variable libre x dans un terme M, notée $\{N/x\}M$, est définie par induction sur M :

$$\begin{aligned} \{N/x\}x &= N \\ \{N/x\}y &= y && \text{si } x \neq y \\ \{N/x\}(\lambda x.\ M) &= \lambda x.\ M \\ \{N/x\}(\lambda y.\ M) &= \lambda y.\ \{N/x\}M && \text{si } y \notin \mathcal{FV}(N) \\ \{N/x\}(M\ M') &= \{N/x\}M\ \{N/x\}M' \end{aligned} \tag{1.1}$$

Cette définition ne capture jamais les variables. Ainsi, le terme donné par $\{y/x\}(\lambda x.\ y)$ n'est pas défini. Mais nous verrons par la suite que pour la grande majorité des besoins, cette restriction de fait n'entraîne pas de perte de généralité.

Remarque 1.15 L'opération de substitution ainsi définie permet d'éviter les *captures de variables*, qui auraient comme conséquence fâcheuse si elles advenaient, de compromettre la confluence de la β-réduction (Barendregt, 1985).

Définition 1.16 (Réduction forte) La réduction forte est la relation définie inductivement comme suit :

$$\frac{}{(\lambda x.\ M)\ N \longrightarrow_\beta \{N/x\}M}\ (\beta) \qquad \frac{M \longrightarrow_\beta M'}{\lambda x.\ M \longrightarrow_\beta \lambda x.\ M'}\ (\zeta)$$

$$\frac{M \longrightarrow_\beta M'}{M\ N \longrightarrow_\beta M'\ N}\ (\mu) \qquad \frac{N \longrightarrow_\beta N'}{M\ N \longrightarrow_\beta M\ N'}\ (\nu)$$

Une autre manière de définir la β-réduction est de donner la règle (β), puis de prendre la clôture par contexte de celle-ci. Nous avons besoin pour cela de la notion de contexte, ce qui nous permet au passage de définir la notion de congruence, utile pour identifier les termes égaux modulo quelques noms de variables. Bien qu'un contexte soit moralement un terme contenant un « trou », nous préférerons une définition inductive à l'envers inspirée par le *zipper* de Huet (1997), où un contexte est un fil d'Ariane retraçant le chemin entre la racine du terme et un trou. Cette trace est un télescopage contenant suffisamment d'information pour reconstruire le terme étant donné une instance pour le trou.

Définition 1.17 (Contexte, contexte faible) Soit $[]$ un symbole qui n'est pas un nom de variable. Alors un contexte est un terme défini inductivement par la grammaire suivante :

$$\mathsf{TermCtx} \quad \ni \quad C, C' \quad ::= \quad [] \mid C[\lambda x.\ []] \mid C[M\ []] \mid C[[]\ N]$$

Le *trou* d'un contexte est la dernière occurrence de $[]$ dans le contexte. Il nous arrivera de considérer des contextes plus restreints où les trous ne peuvent apparaître que dans certaines positions du terme. Nous donnerons alors explicitement la grammaire génératrice de l'ensemble de ces contextes.

Définition 1.18 (Profondeur d'un contexte) La profondeur d'un contexte C est donnée par récurrence sur C :

$$d([]) = 0 \qquad d(C[M\ []]) = 1 + d(C)$$
$$d(C[\lambda x.\ []]) = 1 + d(C) \qquad d(C[[]\ N]) = 1 + d(C)$$

Définition 1.19 (Remplissage) le remplissage d'un contexte C par un terme N, noté $C[N]$, est défini ainsi :

$$[][N] = N \qquad C[M\ []][N] = C[M\ N]$$
$$C[\lambda x.\ []][N] = C[\lambda x.\ N] \qquad C[[]\ M][N] = C[N\ M]$$

Définition 1.20 (Stable par contexte, radical, clôture par contexte, réécriture de tête) Une relation $(\rightharpoonup)$ est *stable par contexte* si pour tous M et M', $M \rightharpoonup M'$ implique $C[M] \rightharpoonup C[M']$ pour tout contexte C . Tout terme M non normal pour $(\rightharpoonup)$ est un *radical*. La *clôture par contexte* est la plus petite relation $(\longrightarrow)$ stable par contexte contenant $(\rightharpoonup)$. Nous dirons de la clôture par le contexte trivial $[]$ qu'elle réécrit les termes *en position de tête*.

Définition 1.21 (Congruence) Une *congruence* est une relation d'équivalence stable par contexte.

Puisque le λ-calcul fut conçu comme un langage de fonctions, il épouse les conventions de nommage des mathématiques. Ainsi le mathématicien écrira indifféremment $f(x) = x^2$ et $f(y) = y^2$ pour exprimer la fonction de mise au carré. Remarquons que dans le λ-calcul le choix particulier du nom de chaque variable n'est pas pertinent pour le calcul : l'ensemble des chemins de réduction possibles pour $(\lambda x.\ (\lambda y.\ y)\ x)\ z$ et pour $(\lambda z.\ (\lambda y.\ y)\ z)\ z$ sont égaux, au renommage de variables près.

Définition 1.22 L'*α-conversion*, notée $(=_\alpha)$, est la plus petite congruence sur les λ-termes telle que $\lambda x.\ M =_\alpha \lambda y.\ \{y/x\}M$ pour tout y tel que $y \notin \mathcal{FV}(M)$.

Tant que l'ensemble des variables est infini, il est facilement démontrable par induction sur un λ-terme M qu'il existe toujours un M' tel que les variables liées dans M n'appartiennent pas à un ensemble donné. On peut donc toujours renommer les variables dans un terme M afin que la condition de bord dans l'équation (1.1) soit bien respectée et la substitution bien définie (Barendregt, 1985). Dans cette thèse, nous identifierons implicitement presque systématiquement tous les termes d'une même classe d'α-équivalence.

De même que le nom des variables n'est pas pertinent pour le calcul, étant donné un terme f, il existe de nombreux termes dont le comportement calculatoire est équivalent à f mais que l'on ne sait pas β-réduire à f sans en savoir plus sur f. Par exemple, $(\lambda x_1.\ f\ x_1)$ aura la même forme normale, ainsi que $(\lambda x_2.\ (\lambda x_1.\ f\ x_1)\ x_2)$, etc. Il est utile alors d'introduire une nouvelle notion d'équivalence.

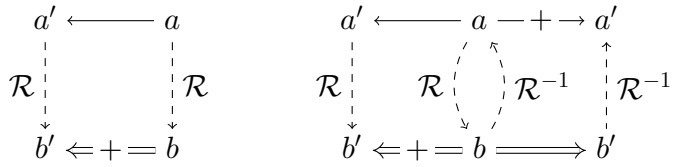
Définition 1.23 La *η-équivalence* est la plus petite congruence sur les λ-termes générée par la règle

$$\lambda x.\ f\ x \longrightarrow_\eta f$$

1.5 Des arbres au code machine

Nous passerons dans cette section d'un système de réécriture à d'autres systèmes de réécriture jusqu'à obtenir des suites de code machine. L'interprétation de ces suites d'instructions peut elle-même être décrite comme un système de réécriture sur les états possible de la machine, où les règles du système de réécriture ne s'appliquent qu'en position de tête. On qualifiera indifféremment de tels systèmes de réécriture de *systèmes de transition d'états* et nous nous intéresserons à la simulation d'un système dans l'autre. Parfois les étapes de réduction ou de transition de deux systèmes se feront de concert, auquel cas ces deux systèmes sont dits *bisimilaires*.

Définition 1.24 (Simulation, bisimulation) Soit A,B équipés chacun d'une relation binaire $(\longrightarrow)$ et $(\Rightarrow)$, respectivement, et $\mathcal{R}$ une relation sur A et B. $\mathcal{R}$ est une relation de *simulation* si et seulement si pour tout a,$a' : A$ et pour tout $b : B$, si $a \longrightarrow a'$ et $a\mathcal{R}b$, alors il existe un $b' : B$ tel que $b \Rightarrow^+ b'$ et $a'\mathcal{R}b'$. $(\longrightarrow)$ et $(\Rightarrow)$ sont *bisimilaires* si $\mathcal{R}$ et $\mathcal{R}^{-1}$ sont toutes deux des relations de simulation. Nous dirons que $(\Rightarrow)$ simule $(\longrightarrow)$ par $\mathcal{R}$.



simulation bisimulation

Étape 1 : des noms aux pointeurs

Le λ-calcul tel que défini ci-dessus permet de manipuler des fonctions en les adressant par leur nom. Bien qu'agréable d'usage pour le logicien aussi bien que pour le programmeur, deux gros utilisateurs de variantes de ce calcul, ce mode d'adressage n'est pas approprié pour une implémentation machine efficace. La substitution d'un terme N sous une abstraction ne peut se faire que sous la condition que la variable que lie l'abstraction n'apparaît pas libre dans N. Remplir cette condition est coûteux car il devient alors nécessaire d'étiqueter chaque terme par l'ensemble de ses variables libres, ou alors de générer cet ensemble à la volée[3]. De plus, au moment du passage de la substitution sous l'abstraction, il se peut qu'il faille renommer la variable liée par l'abstraction afin de se plier à cette condition. Le renommage d'une variable dans un terme est lui aussi coûteux, car supposant de parcourir le corps de l'abstraction entièrement.

Soit $M_1, \ldots, M_n$ un ensemble de termes. Certains seraient tentés de renommer toutes les variables dans chaque M_i afin de postuler que toutes les abstractions lient une variable avec un nom différent de celui lié par toutes les autres abstractions et le nom de toutes les variables libres dans $M_1, \ldots, M_n$ — propriété que l'on appellera *unicité des variables*[4]. Ainsi

[3] Cet étiquetage est une forme de *chaine de définition-utilisation*, structure courante pour supporter l'emploi d'optimisations tel que la propagation de constantes ou l'élimination de sous-expressions communes (Aho et al., 1986 chapitre 9). L'ensemble des variables libres d'un terme M est un sous-ensemble des variables libres de tous les sous-termes de M. On peut donc construire $\mathcal{FV}(M)$ en partageant les ensembles $\mathcal{FV}(M_i)$ associés à chaque sous-terme M_i de M. Ce partage se fait cependant au prix d'un temps d'accès plus longs dans les ensembles de variables libres.

[4] On retrouve parfois cette propriété dans la littérature sous le nom de « convention de Barendregt » bien que celle-ci fut portée à la connaissance de Barendregt en 1972 par Thomas Ottmann (Barendregt, 2004). D'autre part, cette convention, du moins popularisée par Barendregt, est autre :

« If $M_1, ..., M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables. »

(Barendregt, 1985 page 26)

la substitution d'un terme dans un autre est toujours possible sans renommage lorsque ces deux termes vérifient la propriété d'unicité des variables. Malheureusement, cette propriété n'est pas stable par substitution et donc par réduction.

Exemple 25 Soit $M = (\lambda y.\ y\ y)\ (\lambda x.\ x)$ et $M' = (\lambda x.\ x)\ (\lambda x.\ x)$. Nous avons $M \longrightarrow_\beta M'$, bien que M vérifie la propriété d'unicité des variables alors que M' ne la vérifie pas.

L'approche choisie par de Bruijn tôt dans le développement du projet AUTOMATH consiste à s'affranchir entièrement de l'alpha conversion en supprimant les noms des λ-termes (De Bruijn, 1972). Les variables se muent en pointeurs vers l'abstraction qui les a déclaré. Les pointeurs peuvent être représentés textuellement par des numéros.

Définition 26 (Les λ_{dB}-termes) L'ensemble des termes du λ_{dB}-calcul est défini inductivement par la grammaire suivante :

$$\begin{array}{rcll} \mathbb{N} & \ni & k, n, m & \\ \mathsf{Term}_{\mathsf{dB}} & \ni & M, N \quad ::= & n \mid \lambda\ M \mid M\ N \end{array}$$

Nous usons des mêmes conventions d'associativité et de priorité que pour le λ-calcul et transposons les notions de contexte et de remplissage à la syntaxe du λ_{dB}-calcul.

Parmi tous les choix possibles de système de numérotation des pointeurs, nous en utiliserons deux dans cette thèse.

Dans le premier système, appelé *indices de de Bruijn*, le numéro d'un pointeur donne le nombre d'abstractions intervenant entre le pointeur et l'abstraction auquel il se réfère. Le terme $\lambda x.\ \lambda y.\ (\lambda z.\ x\ y)\ y$ s'écrit donc

$$\lambda\ \lambda\ (\lambda\ 2\ 1)\ 0$$

Ce système a l'avantage que la numérotation est stable par contexte.

Les *niveaux de de Bruijn* forment un système de numérotation dual au premier. Ici, un pointeur n se réfère à la $(n-1)^{\text{e}}$ abstraction en traversant le terme de la racine jusqu'au pointeur. Le terme $\lambda x.\ \lambda y.\ (\lambda z.\ x\ y)\ y$ s'écrit donc

$$\lambda\ \lambda\ (\lambda\ 0\ 1)\ 1$$

Ce système a la propriété intéressante que les pointeurs vers la même abstraction ont le même numéro, quel que soit la distance entre les pointeurs et l'abstraction.

Par application du « no free lunch theorem » si récurrent en informatique, passer de variables nommées à un mode d'adressage des abstraction par pointeurs numérotés ne signifie pas pour

autant que la substitution d'un terme dans un autre en devient triviale. L'absence de noms rend l'α-conversion obsolète, mais en contrepartie certains des pointeurs doivent être renumérotés à chaque substitution, que l'on utilise des indices ou des niveaux de de Bruijn.

Nous choisissons pour le reste de ce chapitre une numérotation en indices de de Bruijn. Le remplissage d'un trou dans un contexte C par un λ_{dB}-terme M suppose une opération de renumérotation des indices dans ce terme. En effet, il peut exister des pointeurs dans M qui se réfèrent à des abstractions en dehors de M. Ces pointeurs sont *ballants*. Ils doivent être renumérotés en fonctions du nombre d'abstractions trouvées sur le chemin entre le trou et la racine du contexte C. Sinon ceux-ci ne seraient plus ballants mais pointeraient alors vers les abstractions éventuelles du contexte C — un phénomène très similaire à celui de capture dans le cas du λ-calcul nommé.

Définition 27 (Niveau d'un contexte) Le niveau d'un contexte est le nombre d'abstractions entre la racine du contexte et son trou.

Définition 28 (Grimper dans un λ_{dB}-terme) Les opérations de renumérotation et de substitution définies ci-dessous maintiennent toutes deux un décompte du nombre d'abstractions traversées lors de leur parcours d'un terme à partir de sa racine. Les pointeurs en dessous du niveau courant ne sont pas touchés. Nous définissons donc une fonction générique $(//)$ tenant compte du niveau courant. Elle est paramétrée par deux fonctions $f_=$ et $f_<$, à appliquer selon les cas :

$$ {}^{f_<}_{f_=}\!/\!/_k\ n = \begin{cases} f_= \ k & \text{si } k = n \\ f_< \ k \ n & \text{si } k < n \\ n & \text{sinon} \end{cases} $$

$$ {}^{f_<}_{f_=}\!/\!/_k\ (\lambda\ M) = \lambda\ ({}^{f_<}_{f_=}\!/\!/_{k+1}\ M) $$

$$ {}^{f_<}_{f_=}\!/\!/_k\ (M\ \ N) = ({}^{f_<}_{f_=}\!/\!/_k\ M)\ ({}^{f_<}_{f_=}\!/\!/_k\ N) $$

Définition 29 (Renumérotation d'indices et substitution pour un indice) Soit M,N deux λ_{dB}-termes et n un indice. L'opération de renumérotation $(\uparrow)$ est définie par

$$ \uparrow_k^n M = {}^{f_<}_{f_=}\!/\!/_k\ M \quad \text{où } f_= = \lambda k.\ n + k \text{ et } f_< = \lambda k.\ \lambda n'.\ n + n', $$

et l'opération de substitution $\{N/n\}M$ est définie par

$$ \{N/n\}M = {}^{f_<}_{f_=}\!/\!/_k\ M \quad \text{où } f_= = \lambda k.\ \uparrow_0^k N \text{ et } f_< = \lambda k.\ \lambda n'.\ n' - 1. $$

Il est possible de construire une fonction appelée *interprétation* produisant un λ_{dB}-terme à partir d'un λ-terme tel que cette interprétation est une bijection modulo α-conversion. Chaque étape de réduction du terme initial correspond à une étape de réduction de l'interprétation et *vice versa*. Le λ-calcul et le λ_{dB}-calcul sont donc bisimilaires. Passer d'un calcul nommé

à un calcul d'indice est ainsi une première instance de simulation d'un calcul formel par un autre. C'est aussi souvent une première étape de la traduction d'un λ-terme vers une suite d'instructions haut niveau pour une machine abstraite, puis éventuellement la simulation de ces instructions haut niveau par du code bas-niveau pour un processeur au cœur d'un ordinateur.

Étape 2 : des termes aux clôtures avec les substitution explicites

Le passage des λ-termes aux λ_{dB}-termes a été motivé par un souci d'efficacité. Mais chaque étape de réduction de λ_{dB}-termes telle que définie dans l'étape 1 n'en induit pas moins beaucoup de calculs redondants. Nous présentons dans cette section un cadre formel permettant de raisonner finement sur le coût de chaque substitution. Nous étudierons à l'étape 3 comment fixer une stratégie de réduction et exploiter ce choix afin de réduire les calculs nécessaires à chaque substitution. Nous verrons aussi comment fusionner une partie du travail de réductions successives à l'étape 4.

La β-réduction était jusqu'à présent définie en terme de substitution, elle-même une méta-opération dans le sens ou celle-ci était défini non pas comme un terme, mais comme une fonction du métalangage manipulant des termes. Le λ-calcul de Church est un modèle pratique de toutes les fonctions effectivement calculables mais ne rend pas compte des propriété « opérationnelles » du calcul de ces fonctions, puisque la substitution n'est pas une opération primitive du calcul et son coût peut être arbitrairement élevé en fonction de ses arguments. Curry et Feys (1958) propose une première solution en présentant un modèle syntaxique des fonctions calculables où le coût de chaque étape de réduction est borné. Cependant, les étapes de réduction de la *logique combinatoire* de Curry et Feys peuvent être bien plus nombreuses qu'il n'est strictement nécessaire. Elle ne permet donc pas de donner une borne inférieure au calcul nécessaire à la normalisation d'un terme et ne présente pas un modèle efficace de calcul.

Rappelons qu'en proposant une syntaxe simple avec une sémantique précise, le λ-calcul permet de raisonner finement sur les fonctions. Afin d'analyser le coût calculatoire de chaque réduction, et donc de chaque substitution, l'idée d'un calcul avec substitutions explicites est d'internaliser la substitution dans la syntaxe des termes. La substitution est promue de la sorte comme une citoyenne de première classe, au même titre que les fonctions dans le λ-calcul. La syntaxe du calcul obtenu n'en demeure pas moins simple et la sémantique tout aussi précise, de sorte qu'il devient possible de raisonner finement, non seulement sur les fonctions, mais sur le coût calculatoire des substitutions.

L'idée d'internaliser les substitutions est d'abord apparue comme une astuce d'implémentation dans le projet AUTOMATH (De Bruijn, 1978). L'idée fut ensuite popularisée par Abadi et al. (1991), qui proposent le λ_σ-calcul comme une reformulation du λ-calcul avec des substitutions

explicites, bien que des travaux antérieurs introduisent déjà des constructions similaires (Revesz, 1985 et Hardin et Lévy, 1989). Nous utilisons ici une présentation inspirée de (Abel, 2010).

Définition 30 (λ_σ-termes) Les termes du $\lambda\sigma$-calcul sont donnés inductivement par la grammaire suivante :

$$\begin{array}{rcccl} \mathsf{Term}_\sigma & \ni & M, N & ::= & 0 \mid \lambda\ M \mid M\ N \mid \sigma\ M \\ \mathsf{Subst} & \ni & \sigma, \tau & ::= & \uparrow \mid \mathsf{id} \mid (\sigma, M) \mid \sigma\ \tau \end{array}$$

Le pointeur d'indice 0 est noté ici par le terme 0. Les pointeurs d'indice n sont représentés par n applications de $\uparrow$ à 0. La substitution id est la substitution qui à tout terme M associe M. L'extension (id, M) de l'identité par M est notée $[M]$.

Bien des applications du λ-calcul excluent toute réduction sous une abstraction (la réduction est dite *faible*[5]). En programmation par exemple, on ne calcule avec une fonction que lorsque celle-ci est appliquée à un argument. Nous verrons plus tard que cette restriction du contexte de réduction permet aussi de rendre chaque étape de réduction moins coûteuse. Nous distinguons donc la réduction faible de la réduction forte dans les règles de calcul du λ_σ-calcul.

Définition 31 (Réduction forte pour le λ_σ-calcul) Soit l'ensemble de règles suivant[6] :

$$(\lambda\ M)\ N \rightharpoonup [N]\ M \qquad (2)$$

$$(\sigma, M)\ 0 \rightharpoonup M$$

$$(\sigma, M)\ \uparrow \rightharpoonup \sigma$$

$$\mathsf{id}\ M \rightharpoonup M$$

$$\sigma\ (\lambda\ M) \rightharpoonup \lambda\ (\uparrow\ \sigma, 0)\ M \qquad (3)$$

$$\sigma\ (M\ N) \rightharpoonup (\sigma\ M)\ (\sigma\ N)$$

$$\tau\ (\sigma\ M) \rightharpoonup (\tau\ \sigma)\ M \qquad (4)$$

$$(\tau\ \sigma)\ M \rightharpoonup \tau\ (\sigma\ M) \qquad (5)$$

$$\tau\ (\sigma, M) \rightharpoonup (\tau\ \sigma, \tau\ M) \qquad (6)$$

La règle (2) exprime la β-réduction. Les autres règles propagent les substitutions à travers les branches d'un terme M. L'application successive de deux substitutions σ,τ à M se réduit en l'application d'une seule substitution, donnée par la composition de σ et τ. La règle (6) exprime ce que signifie composer deux substitutions : $\tau\ \sigma$ est la substitution où τ a été appliquée à chaque élément de σ. La règle (3) permet la propagation d'une substitution σ sous une abstraction. Il faut alors réhausser tous les indices dans σ et subsitutier l'indice 0 par lui-même.

[5] À ne pas confondre avec la propriété de normalisation faible.

[6] Certains termes normalisants du λ-calcul ne sont que faiblement normalisants dans le λ_σ-calcul à cause des règles (4) et (5). Ces règles sont néanmoins nécessaires pour la confluence du calcul ainsi que la simulation du λ-calcul, et l'absence de cette propriété de *préservation de la normalisation* n'est pas un problème pour la suite de ce chapitre.

À cet ensemble de règles sont ajoutées les règles non-calculatoires suivantes, afin d'assurer la confluence locale du λ_σ-calcul :

$$(\uparrow, 0) \rightharpoonup \mathsf{id} \qquad\qquad \mathsf{id}\ \sigma \rightharpoonup \sigma$$

$$\sigma_3\ (\sigma_2\ \sigma_1) \rightharpoonup (\sigma_3\ \sigma_2)\ \sigma_1 \qquad\qquad \sigma\ \mathsf{id} \rightharpoonup \sigma$$

La relation de réduction $(\longrightarrow_\sigma^{\mathsf{nf}})$ est la clôture par contexte de la relation induite par les deux ensembles de règles ci-dessus.

Définition 32 (Réduction faible pour le λ_σ-calcul) Une relation de réduction $(\longrightarrow_\sigma)$ est la clôture par le contexte C de la relation $(\rightharpoonup)$ engendrée par les règles de la réduction forte ou l'équation (3) est remplacée par :

$$(\sigma\ (\lambda\ M))\ N \rightharpoonup (\lambda\ (\sigma\ \uparrow, 0)\ M)\ N$$

Ainsi une substitution n'est propagée dans le corps d'une abstraction que si celle-ci est appliquée.

Dans tous les langages précédents, les substitutions étaient formées de couples d'une variable et d'un terme. Les substitutions étaient appliquées une par une de sorte que le terme dans lequel se faisaient les substitutions était traversé plusieurs fois. Le λ_σ-calcul généralise ces substitutions unaires à des substitutions n-aires en permettant de combiner les substitutions, de telle sorte que les substitutions ne sont plus seulement des couples mais des listes de couples. Une substitution est construite par extension d'une substitution déjà existante ou par concaténation de deux substitutions. Ce chaînage des couples permet de les propager dans le terme et de les appliquer simultanément, réduisant ainsi le nombre de parcours dans le terme.

Étape 3 : fixer une stratégie de réduction

La relation de réduction définie pour le λ-calcul à l'aide d'une clôture par contexte de la règle (β) permet d'atteindre la forme normale d'un terme par plus d'un chemin, si cette forme normale existe. Même la restriction du contexte de réduction à l'extérieur des abstractions ne fait pas de la relation de réduction une fonction. Ce non déterminisme rend la tâche du programmeur délicate car il devient difficile de prévoir le comportement calculatoire d'un terme. Certains chemins de réduction de $(\lambda x.\ y)\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))$ sont infinis alors que d'autres sont de taille finie. De plus, deux chemins de réductions d'un terme même fortement normalisant peuvent manifester deux profils en mémoire et en temps de calcul très différents.

Imaginons l'existence d'un oracle qui pour toute relation $(\longrightarrow)$ et tout terme M non normal pour $(\longrightarrow)$ donne le choix du radical à réduire, tel que des appels itérés à l'oracle permettront

de suivre le chemin de réduction le plus court vers la forme normale de M. Cet oracle comme tant d'autres est une chimère, mais on connaît depuis Lévy (1978) un algorithme permettant de simuler les réponses qu'aurait donné cet oracle. En pratique cependant, cet algorithme est tellement coûteux en ressources de calcul que les implémentations usuelles obtiennent souvent de meilleurs résultats en s'en remettant à des fonctions heuristiques, que l'on appellera *stratégies* de réduction.

Définition 33 (Ordre normal de réduction) La réduction en ordre normal est la clôture par le contexte suivant de la règle (β) :

$$C \quad ::= \quad C\ M \mid []$$

Cette stratégie de réduction est appellée évaluation en « appel par nom ».

Théorème 34 *L'appel par nom est une stratégie complète.*

Définition 35 (Valeur) L'ensemble des valeurs est le sous-ensemble des termes donné par la grammaire suivante :

$$\mathsf{Term}^{\mathsf{V}} \quad \ni \quad v, v_i \quad ::= \quad x \mid \lambda x.\ M \mid x\ v_1 \ldots v_n$$

Définition 36 (Ordre applicatif de réduction) Soit (β_v) la règle de réduction suivante :

$$\frac{}{(\lambda x.\ M)\ v \longrightarrow_\beta \{v/x\}M}\ (\beta_v)$$

La réduction en ordre applicatif est la clôture par le contexte suivant de la règle (β_v) :

$$C \quad ::= \quad C\ M \mid M\ C \mid []$$

Cette stratégie de réduction est appellée évaluation en « appel par valeur ».

Il se peut que la réduction en appel par valeur n'atteigne jamais la forme normale d'un terme, mais l'appel par nom entraîne souvent la duplication inutile de radicaux, comme dans l'exemple suivant :

$$\begin{aligned}(\lambda x.\ x\ x)\ (\underline{(\lambda y.\ y)\ (\lambda z.\ z)}) &\longrightarrow (\underline{(\lambda y.\ y)\ (\lambda z.\ z)})\ (\underline{(\lambda y.\ y)\ (\lambda z.\ z)}) \\ &\longrightarrow (\lambda z.\ z)\ ((\lambda y.\ y)\ (\lambda z.\ z)) \\ &\longrightarrow ((\lambda z.\ z)\ (\lambda z.\ z)) \\ &\longrightarrow (\lambda z.\ z)\end{aligned}$$

Les radicaux présents dans l'argument d'une fonction (souligné) seront répétés autant de fois que la variable liée à la tête de la fonction est répété dans le corps de celle-ci. Dans la pratique, les implémentations de variantes du λ-calcul adoptant une stratégie d'appel par nom partagent chaque instance de l'argument d'une fonction. Moralement, les termes sont représentés par des graphes plutôt que comme des arbres. La réduction d'un noeud est ainsi partagée par tous ses parents. Cette stratégie réduit les arguments plus tard qu'en appel par valeur tout en évitant de nombreuses duplications ; on la qualifie donc d'évaluation *paresseuse*. Pour tout terme M, le nombre de réductions lors de l'évaluation paresseuse de M est inférieur ou égal à son évaluation en appel par valeur. Mais à l'instar de la réduction optimale de Lévy, le coût administratif de la maintenance du graphe nécessaire à l'évaluation paresseuse anéantit souvent les gains dûs au nombre de réductions plus faible.

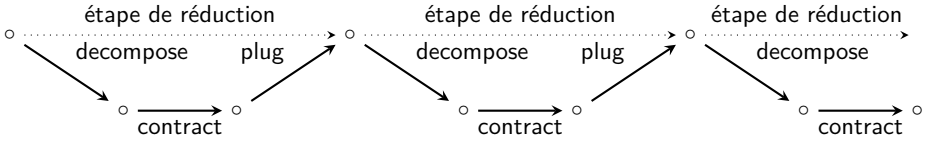
Étape 4 : machines abstraites

Étape 4.1 : Refocalisation du calcul

Fixer un terme M et une stratégie de réduction suffit à spécifier un calcul déterministe qui opérera par itération jusqu'à atteindre un point fixe ou diverger de

a. une recherche d'un radical dans un terme (decompose),
b. suivie d'une contraction de ce radical (contract),
c. et enfin du remplissage de la contraction dans son contexte (plug).

Puisque le choix de la stratégie de réduction fixe le choix du radical à contracter s'il existe, decompose est une fonction, comme le sont aussi contract et plug. Schématiquement, le calcul procède ainsi :



Chacune de ces étapes peuvent être exprimées à l'aide d'un ensemble de règles de réduction qui ne réécrivent qu'en position de tête. Ces règles peuvent être combinées pour former un système de transition d'états de S, comme pour le système suivant pour la réduction faible en appel par nom.

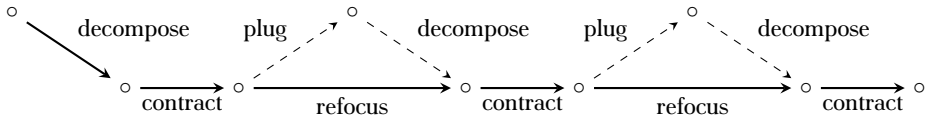
Exemple 37 Soit un ensemble de symboles de phase $\{\mathsf{decompose}, \mathsf{contract}, \mathsf{plug}, \mathsf{value}\}$. Soit S le type des états formés de tuples d'un symbole de phase, d'un terme et d'un contexte. Le

système de transition d'états dont les règles de transition sont données ci-dessous simule la réduction faible de termes du λ_σ-calcul.

$$\begin{aligned}
\langle \mathsf{decompose}, r, C\rangle &\Rightarrow \langle \mathsf{contract}, r, C\rangle \\
\langle \mathsf{decompose}, v, []\rangle &\Rightarrow \langle \mathsf{value}, v, []\rangle \\
\langle \mathsf{decompose}, v, C[[]\ M]\rangle &\Rightarrow \langle \mathsf{value}, v\ M, C\rangle \\
\langle \mathsf{decompose}, M\ N, C[[]\ M]\rangle &\Rightarrow \langle \mathsf{decompose}, M, C[[]\ N]\rangle \\
\langle \mathsf{contract}, r, C\rangle &\Rightarrow \langle \mathsf{plug}, r', C\rangle \qquad \text{si } r \rightharpoonup r' \\
\langle \mathsf{plug}, M, C\rangle &\Rightarrow \langle \mathsf{decompose}, C[M], []\rangle
\end{aligned}$$

où r est un radical (c'est-à-dire de l'une des formes du membre gauche des règles de la définition 32), et v est une valeur.

Ce schéma de calcul est très inefficace. Bien que la contraction d'un radical puisse en créer de nouveaux, ces nouveaux radicaux ne peuvent apparaître que localement dans le terme. Recommencer la recherche de radical depuis la racine du terme est donc inefficace. Danvy et Nielsen (2004) proposent une transformation mécanique de nombreux calculs spécifiés à l'aide des trois composantes decompose, contract et plug. Les phases de décomposition et de remplissage peuvent être remplacées par une seule phase dite de *refocalisation* (ou « refocusing » en anglais).



La refocalisation est une simple généralisation de la décomposition d'un terme en un radical et son contexte à la décomposition d'un terme dans un contexte quelconque en un radical et son contexte.

Si l'on écrit decompose et plug comme des fonctions, Danvy et Nielsen (2004) et Danvy (2009) observent que

$$\mathsf{decompose}(\mathsf{plug}(M, C), []) = \mathsf{decompose}(M, C)$$

pour tout terme M et contexte C. On peut recommencer en prenant le contexte C comme point de départ plutôt que la racine du terme obtenu par remplissage de M dans C. La phase plug n'est donc plus nécessaire. En fusionnent les phases decompose et contract en une seule, le système obtenu est le suivant :

$$\begin{aligned} \langle r, C\rangle &\Rightarrow \langle r', C\rangle \qquad \text{si } r \rightharpoonup r' \\ \langle v, C[[]M]\rangle &\Rightarrow \langle v\ M, C\rangle \\ \langle M\ N, C\rangle &\Rightarrow \langle M, C[[]\ N]\rangle \end{aligned} \tag{7}$$

Un système de transition d'états de ce style est aussi appelé *machine abstraite*.

Remarque 38 Il est aussi possible de trouver la valeur d'un terme par une fonction récursive, ou *fonction d'évaluation*. La qualité distinctive d'une machine abstraite est qu'il s'agit d'un processus itératif, et non récursif. Une présentation itérative peut être avantageuse car le flot de contrôle d'une fonction récursive devient alors explicite, ce qui facilite les analyses et les optimisations.

La machine de Krivine

La machine de Krivine (1985,2007) est une instance de machine abstraite calculant la forme normale de tête faible d'un terme en optant pour une stratégie en appel par nom. Les stratégies ne réduisant jamais sous les abstractions risquent de dupliquer des radicaux inutilement mais en contrepartie les réductions s'en trouvent simplifiées puisque les substitutions n'ont jamais besoin d'être renumérotées si celle-ci ne traversent jamais une abstraction. Remarquons en particulier que pour tous termes M,N et toute substitution σ, la séquence de réduction

$$(\sigma\ (\lambda\ M))\ N \longrightarrow_\sigma (\lambda\ (\sigma\ \uparrow, 0)\ M)\ N \longrightarrow_\sigma [N]\ ((\sigma\ \uparrow, 0)\ M) \longrightarrow_\sigma^* (\sigma, N)\ M$$

est une simulation de l'application de la règle suivante

$$(\sigma\ (\lambda\ M))\ N \longrightarrow_\sigma (\sigma, N)\ M \tag{8}$$

Ainsi les règles (2) et (3) pourraient être remplacées par la règle (8) qui n'opère pas de renumérotation des indices dans les substitutions ou dans les termes. Nous pouvons par ailleurs supprimer les règles (4) de concaténation de substitutions, dans la mesure où la nouvelle règle (8) combine directement les substitutions.

La machine de Krivine part d'un calcul de substitutions explicites dans lequel la β-réduction du λ-calcul originel est simulé par la règle (8). Les termes dans ce genre de calcul ont parfois une substitution associée, bien que ce ne soit pas systématique. La machine de Krivine quant à elle considère que tous les termes sont systématiquement associés à une substitution, ne serait-ce que l'identité. Elle ne manipule donc pas directement des termes mais plutôt une construction, due à Hasenjaeger et Scholz (1961) en logique et à Landin (1964) en informatique, nommée

clôture : une paire d'un terme M et d'une substitution associant un terme à toutes les variables libres de M.

Définition 39 (La machine de Krivine) Soit les listes de paires définies inductivement comme suit :

$$E, E_n, S \quad ::= \quad [] \mid (E_1, M_1) :: E$$

La machine de Krivine[7] est un système de transition d'états dont les règles de transitions sont les suivantes :

$$\langle 0, (E_1, M_1) :: E, S\rangle \Rightarrow \langle M_1, E_1, S\rangle$$
$$\langle \uparrow M, (E_1, M_1) :: E, S\rangle \Rightarrow \langle M, E, S\rangle$$
$$\langle \lambda\, M, E, (E_1, M_1) :: S\rangle \Rightarrow \langle M, (E_1, M_1) :: E, S\rangle$$
$$\langle M\ N, E, S\rangle \Rightarrow \langle M, E, (E, N) :: S\rangle$$

Le deuxième élément du tuple d'un état est *l'environnement* de la machine. Le troisième élément est la *pile d'arguments*.

L'ensemble de tuples qu'est l'ensemble des états de la machine de Krivine comprend un terme et sa substitution associée, ainsi qu'une pile d'arguments auxquels le terme est appliqué. Bien qu'elle ait été inventée indépendemment, cette machine n'a rien de « magique » dans la mesure où elle est le résultat d'une transformation systématique et entièrement mécanisable (Biernacka et Danvy, 2007) de la machine abstraite obtenue à la fin de la section précédente. En particulier, puisqu'une propagation d'une substitution à travers une application est toujours suivie d'une réécriture par la règle (7), la séquence de réduction

$$\langle \sigma\ (M\ N), C\rangle \Rightarrow \langle (\sigma\ M)\ (\sigma\ N), C\rangle \Rightarrow \langle (\sigma\ M), C[[](\sigma\ N)]\rangle$$

peut être internalisée par une règle

$$\langle \sigma\ (M\ N), C\rangle \Rightarrow \langle (\sigma\ M), C[[]\ (\sigma\ N)]\rangle$$

qui remplace les règles de propagation à travers une application et la règle (7). Décomposer la clôture dans le premier élément du tuple en son terme et sa substitution associée donne la dernière règle de transition de la machine de Krivine ci-dessus, où le contexte modélise une pile.

[7] La présentation donnée ici n'est pas la version originelle de la machine de Krivine mais basée sur une reconstruction rationelle motivée par certaines formulations catégoriques de la sémantique du λ-calcul (Selinger, 2003).

La très grande majorité des ordinateurs actuels utilisent une architecture en pyramide de la mémoire servant à stocker les programmes à exécuter ainsi que leurs données. Une petite partie de la mémoire sera très rapide mais très coûteuse, une bien plus large partie sera plus lente mais moins coûteuse, et ainsi de suite. Pour optimiser les performances, les processeurs de ces ordinateurs cherchent à stocker les données les plus régulièrement utilisées par le calcul courant dans la mémoire la plus rapide (que l'on appellera « cache »), de sorte que les accès mémoires fassent perdre moins de temps. Dans le cas général, il n'est pas possible pour le processeur de deviner à l'avance la répartition optimale des données entre la cache et les mémoires plus lentes étant donné les contraintes de taille de chacune. Ceux-ci s'appuient donc sur des fonctions heuristiques cherchant à faire le bon choix pour les calculs les plus courants. Il est observé que pour la plupart des programmes, les accès mémoire durant les cycles d'horloge qui suivent un premier accès mémoire se font presque tous vers des adresses logiques proches. Les processeurs cherchent donc à exploiter la *localité* des accès mémoire en chargeant à titre spéculatif les données à l'adresse $n+1, \ldots, n+m$ dans la cache après un accès à l'adresse n.

Pour améliorer la localité et ainsi améliorer le coût moyen des accès en mémoire, il est donc avantageux de représenter un terme sous la forme d'un bloc contigu en mémoire. Un arbre peut-être représenté de la sorte si c'est une structure invariante dans le temps mais par souci de clarté nous passons à une présentation plus linéaire des termes : une liste d'instructions (ou « bytecode » en anglais). Cette liste pourra être représentée dans une implémentation réelle comme un tableau d'instructions que la machine abstraite parcourt à l'aide d'un pointeur indiquant l'instruction courante dans le tableau à exécuter. On appelle *machine virtuelle* une machine abstraite qui opère sur une suite d'instruction plutôt qu'un arbre.

Définition 40 (Compilation bytecode du λ_σ-calcul) Soit la grammaire suivante de suites d'instructions :

$$c, c', c_n \quad ::= \quad \texttt{access}\ n; c \mid \texttt{grab}; c \mid \texttt{push}\ c'; c \mid \texttt{noop}$$

La compilation des termes du λ_σ-calcul est donnée par :

$$\begin{aligned} [\![\uparrow \cdots \uparrow 0]\!] &= \texttt{access}\ n; \texttt{noop} \quad \text{où } n \text{ est le nombre de } (\uparrow). \\ [\![\lambda\ M]\!] &= \texttt{grab}; [\![M]\!] \\ [\![M\ \ N]\!] &= \texttt{push}\ [\![N]\!]; [\![M]\!] \end{aligned}$$

Définition 41 La machine de Krivine pour des termes compilés est un système de transition d'états dont les règles de transition sont les suivantes :

```
        cmpq    %r8, %r9        ; déterminer si pile est vide.
        jne     .L2             ; si non, continuer à L2.
        movl    $0, %edi
        call    control_halt    ; terminer l'éxécution.
.L2:
        movq    %r10, 24(%r9)   ; déplacer clôture du haut de
                                ; l'environnement à la pile.
        movq    %r9, %r10       ; pointeur d'environnement
                                ; vers nouveau premier élément.
        leaq    -32(%r9), %r9   ; décrémenter pointeur de pile.
```

Figure 1 Instructions en assembleur pour l'architecture x86-64 (syntaxe AT&T) simulant l'instruction `grab` de la machine de Krivine.[8]

$$\langle \texttt{access}\ n; c, E, S\rangle \Rightarrow \langle c, E(n), S\rangle$$
$$\langle \texttt{grab}; c, E, (E_1, c_1) :: S\rangle \Rightarrow \langle c, (E_1, c_1) :: E, S\rangle$$
$$\langle \texttt{push}\ c_1; c, E, S\rangle \Rightarrow \langle c, E, (E, c_1) :: S\rangle$$

où $E(n)$ est une notation pour le $(n-1)^{\text{e}}$ élément de E, si celui-ci existe.

Étape 5 : de l'interprétation à la compilation

L'intérêt d'une machine virtuelle réside en sa capacité à capturer l'essence du calcul à accomplir sans les inefficacités inhérentes à une présentation plus haut niveau, qui elle se veut plus proche des intuitions de l'Homme que des détails techniques des machines réelles. L'essence du calcul que distille la machine virtuelle est une séquence d'instructions manipulant les données du calcul de la manière la plus efficace possible (par exemple sans renumérotation futile d'indices). Mais pour autant ces instructions ne sont pas atomiques, au sens où dans une simulation de cette machine abstraite sur une machine réelle, l'exécution de chaque instruction de la machine virtuelle correspondra à l'exécution de nombreuses instructions de la machine réelle.

À valeur d'exemple, dans le cas précis de la machine de Krivine, l'instruction `grab` sera typiquement implémentée par la séquence d'instructions donnée dans la figure 1. Ici, nous utilisons le registre `r8` pour stocker le début de la zone mémoire où se trouve la pile, et les registres `r9` et `r10` pour stocker les pointeurs vers le haut de la pile et le début de l'environnement,

[8] Le code assembleur ci-dessus est extrait du résultat de la compilation d'un programme en C implémentant la machine de Krivine (Boespflug, 2010).

respectivement. Il est important de stocker ces valeurs dans des registres sur le processeur car elles sont utilisées fréquemment durant le calcul.

Une simulation naïve de l'exécution d'une machine virtuelle sur une machine réelle consiste à associer un bloc *natif* de code à `access` et `push` comme pour l'instruction `grab`, puis d'écrire un bloc de code spécial (*l'interpréteur*) à l'aide duquel on implémente l'algorithme suivant :

1. l'interpréteur se charge de décoder chaque instruction dans la suite d'instruction représentant le terme à évaluer ;
2. celui-ci invoque le bloc de code correspondant à l'implémentation de l'instruction décodée ;
3. lorsque ce bloc de code a terminé le contrôle repasse à l'interpréteur qui recommence à l'étape 1, ou termine l'exécution s'il ne reste plus d'instructions ou si la pile est vide.

Le flot de contrôle de cette simulation saute donc de manière répétée d'un bloc de code à un autre, ceux-ci se trouvant en général à des endroits disparates en mémoire. Ces sauts sont particulièrement dommageables pour la performance de la simulation sur les machines actuelles, car les processeurs actuels font un usage intensif du préchargement des instructions dans une séquence pour améliorer les performances. Chaque saut dans le flot de contrôle rend ce préchargement caduc et a donc un impact très négatif.

La solution à ce problème consiste à remplacer chaque instruction dans la séquence représentant un terme par le bloc de code qui lui correspond et coller tous les blocs de code bout à bout. On obtient par ce procédé un bloc de code bien plus gros que les quatre blocs de code nécessaires précédemment, mais dont l'exécution est plus efficace car le décodage des instructions est éliminé et le nombre de sauts nécessaires dans le calcul est diminué. Ce procédé est qualifié de *compilation native*.

De manière abstraite, une fonction de compilation est une fonction de traduction comme une autre : elle permet de passer d'un calcul exprimé dans un langage L_1 à un autre calcul équivalent exprimé dans un autre langage L_2 en traduisant chaque forme syntaxique de L_1 par sa sémantique dans L_2[9]. L'interprétation du début de section n'est pas une fonction de traduction ; elle forme donc une autre manière de simuler un calcul. Bien que conceptuellement plus complexe et moins performante en pratique, l'interprétation reste néanmoins souvent plus facile à mettre en oeuvre (le calcul peut être accompli directement sans traductions successives dans des langages de plus en plus bas-niveau). Il est également possible d'implémenter directement

[9] Cette vision considère qu'une fonction de compilation est une fonction compositionnelle, c'est à dire définie par induction sur les termes de L_1. Cette condition n'est pas forcément vérifiée dans la pratique — c'est notamment le cas de compilateurs optimisants de la suite — mais nous l'imposons ici par souci de simplicité.

les règles de réduction du λ_σ-calcul sous la forme d'un interpréteur distribuant le contrôle à des blocs de code natifs à tour de rôle. Bien qu'inefficace, cet interpréteur peut être plus performant qu'un compilateur pour des cas particuliers car à la différence de ce dernier, un interpréteur n'a nullement besoin de fixer au préalable un stratégie d'évaluation comme le fait le compilateur pour optimiser les représentations du calcul. Il peut donc changer dynamiquement de stratégie d'évaluation au cours des réductions successives d'un terme pour éviter les duplications de radicaux inutiles (nous y reviendrons lors du chapitre 3).

1.6 Extensions et optimisations

Chacune des phases successive d'un compilateur sont autant de transformations. Les compilateur dits *optimisants* en profitent pour rajouter des transformations additionelles. Certaines de ces transformations sont des *optimisations* : le remplacement d'un programme par un autre sémantiquement équivalent mais arrivant au résultat en moins d'étapes de calcul. Un exemple d'optimisation qui confère un avantage en performance plus grand encore à la compiltation par rapport à l'interprétation est *l'optimisation à lucarne*[10]. En effet, une fois tous les blocs de code mis bout à bout, le compilateur peut reconnaître certaines sous-séquences redondantes et les éliminer ou les réécrire, par exemple

```
incl %eax
addl %ebx, %ecx
decl %eax
```

qui incrémente le registre `eax` pour le décrémenter presque immédiatement. D'autres transformations simplifient les *analyses*, c'est à dire l'extraction d'information du programme, tel que le flot de contrôle ou l'ordre d'évaluation des sous-termes, pour permettre des optimisations plus haut-niveau et globales que l'optimisation à lucarne. Nous traitons ici de la transformation en forme A-normale et de la conversion de clôture, que nous utiliserons durant la vérification de preuves dans la section 4.7.3 et la section 4.8 du chapitre 4.

1.6.1 Le calcul monadique

Nous avons vu que le λ-calcul est un calcul non déterministe, dans la mesure où il peut arriver que plusieurs réductions sont possibles sur un terme. Fixer une stratégie de réduction rend le calcul déterministe. La transformation d'un terme en forme monadique (Moggi, 1989) vise à expliciter cette stratégie dans le terme lui-même, en donnant un nom à chaque résultat

[10] « peephole optimization » en anglais.

intermédiaire d'une réduction. Moggi exprime le λ-calcul monadique à l'aide de deux primitives unit et bind et propose un système de types pour contraindre les termes dans une forme où tous les sous-termes d'un terme sont nommés. Mais il est aussi possible de caractériser les termes en forme monadique syntaxiquement.

Définition 1.42 (λ-calcul monadique) Soit un λ-terme M. M est en *forme monadique* si il est de la forme donnée par la grammaire suivante :

$$\begin{array}{rcccl} \mathsf{Atom} & \ni & a, b & ::= & x \\ \mathsf{Value} & \ni & v, w & ::= & a \mid \lambda x.\ M \\ \mathsf{Term}^{\mathsf{M}} & \ni & M, N & ::= & v \mid a\ b \mid \mathbf{let}\ x \Leftarrow N\ \mathbf{in}\ M \end{array}$$

Dans le λ-calcul pur, seules les variables sont des atomes. Mais des primitives supplémentaires tels que des nombres entiers ou des constantes sont aussi des atomes. La forme $\mathbf{let}\ x \Leftarrow N\ \mathbf{in}\ M$ peut être simulée avec un terme de la forme $(\lambda x.\ M)\ N$. Une alternative consiste à donner la règle de réduction suivante :

$$\mathbf{let}\ x \Leftarrow N\ \mathbf{in}\ M \longrightarrow \{N/x\}M$$

Définition 1.43 (Transformation en forme monadique) La transformation en forme monadique d'un terme est fonction de la stratégie d'évaluation choisie. Voici par exemple une fonction de traduction pour le λ-calcul en appel par valeur :

$$\begin{aligned} \mathcal{E}_v(x) &= x \\ \mathcal{E}_v(\lambda x.\ M) &= \lambda x.\ \mathcal{E}_v(M) \\ \mathcal{E}_v(M\ N) &= \mathbf{let}\ x \Leftarrow N\ \mathbf{in}\ \mathbf{let}\ y \Leftarrow M\ \mathbf{in}\ x\ y \quad \text{où } x \text{ et } y \text{ sont frais} \end{aligned}$$

Définition 1.44 (Forme A-normale) Il est pratique de définir une relation d'équivalence sur les termes en forme monadique donnée par la règle suivante :

$$\mathbf{let}\ x \Leftarrow \mathbf{let}\ y \Leftarrow N\ \mathbf{in}\ M\ \mathbf{in}\ M' \equiv \mathbf{let}\ y \Leftarrow N\ \mathbf{in}\ \mathbf{let}\ x \Leftarrow M\ \mathbf{in}\ M'$$

Cette règle est un exemple de *conversion commutative*. Si nous orientons cette équivalence vers la droite, nous obtenons une règle de réduction. Un terme en forme *A-normale* (Flanagan et al., 1993) est un terme en forme monadique normale pour cette règle de commutation.

La transformation des termes du λ-calcul en forme A-normale est un préalable à la plupart des optimisations dans de nombreux compilateurs, qui l'utilisent comme leur langage intermédaire de choix pour de nombreuses passes d'optimisation (Appel et MacQueen, 1991). Elle est

cousine d'autres langages intermédiaires tels que le langage des termes en style CPS et en forme SSA (Chakravarty et al., 2004) (plus fréquemment utilisée dans les compilateurs de langages impératifs) dans le sens ou ces formes sont presque toujours interdérivables (Appel, 1998 et Kelsey, 1995). Bien que popularisée par Flanagan et al. (1993), remarquons que nous retrouvons la forme A-normale comme résultat de la deuxième étape de la transformation en forme CPS en trois étapes donnée par (Danvy, 1991). Par ailleurs, Hatcliff et Danvy (1994) décomposent la transformation CPS en une transformation dans le calcul monadique de Moggi suivie d'une traduction vers des continuations explicites. Il montrent que le calcul monadique et les termes en CPS sont en correspondance. Certains, tels MLton (Fluet et Weeks, 2001), utilisent un langage intermédiaire combinant ces trois formes.

1.6.2 Conversion de clôture

Le code qu'éxécute la machine de Krivine est toujours clos par un environnement courant. Cet environnement est une structure inductive, où les accès au n-ième élément entraine n transitions de la machine. Représenter l'environnement courant sous forme de tableau est plus avantageux pour la performance de la machine, car alors les accès dans l'environnement peuvent se faire en temps constant.

Mais si les environnements sont des tableaux, alors l'éxécution de l'instruction `push` demande de faire une copie complète de l'environnement courant lorsque celui-ci est poussé sur la pile. Or l'environnement capturé et copié sur la pile par l'instruction `push` n'est pas minimal, au sens où le code clos par cet environnement ne fait pas forcément référence à tous les éléments de l'environnement. Ainsi l'environnement courant est une surapproximation de l'environnement minimal nécessaire pour clôre le code courant. Cette surapproximation entraîne une perte d'espace mémoire et peut empêcher la libération de la mémoire associée à des valeurs qui ne sont plus accessible par le code courant ni par aucun bloc de code sur la pile ou l'environnement.

La conversion de clôture (Steele, 1978) synthétise l'ensemble de variables libres de chaque sous-terme, puis étiquette chaque sous-terme par un environnement minimal explicite sur la base de cette information. Si l'on représente les environnements eux-mêmes par des termes du λ-calcul, alors la conversion de clôture est une transformation terme à terme, comme l'est la transformation en forme monadique de la section précédente.

Plutôt que d'étendre la syntaxe du λ-calcul avec des formes spécifiques pour représenter des environnements ou employer un encodage de Church de ceux-ci, la conversion de clôture donnée ci-dessous représente chaque élément de l'environnement minimal de chaque sous-terme par des abstractions supplémentaires.

Définition 1.45 (Conversion de clôture) Nous définissons la conversion de clôture de termes en forme A-normale comme la fonction suivante, où ρ associe un terme à un nom :

$$\begin{aligned}
\mathcal{C}(x)_\rho &= \rho(x) \\
\mathcal{C}(\lambda x.\ M)_\rho &= \lambda x.\ \mathcal{C}(M)_{\rho[x \mapsto x]} \\
\mathcal{C}(x\ y)_\rho &= \mathcal{C}(x)_\rho\ \mathcal{C}(y)_\rho \\
\mathcal{C}(\mathbf{let}\ x \Leftarrow N\ \mathbf{in}\ M)_\rho &= \mathbf{let}\ x \Leftarrow \lambda y_1.\cdots\lambda y_n.\ \mathcal{C}(N)\ \mathbf{in}\ \mathcal{C}(M)_{\rho'} \\
&\qquad \text{où } \mathcal{FV}(N) = y_1, \ldots, y_n \\
&\qquad \text{et } \rho' = \rho[x \mapsto x\ y_1\ \ldots\ y_n]
\end{aligned}$$

Une fois les variables libres de tous les sous-termes closes par une contexte, il devient possible de hisser n'importe quel sous terme à la racine du terme, sans craindre un quelconque problème de capture. On obtient alors un terme plat, dont toutes les formes $\mathbf{let}\ x \Leftarrow N\ \mathbf{in}\ M$ sont à la racine.

Exemple 1.46 Le terme suivant,

$$\lambda x.\ \lambda y.\ (\lambda z.\ y)\ (\lambda z.\ y\ x)$$

devient, après transformation en forme A-normale,

$$\lambda x.\ \lambda y.\ \mathbf{let}\ a \Leftarrow \lambda z.\ y\ \mathbf{in}\ \mathbf{let}\ b \Leftarrow \lambda z.\ y\ x\ \mathbf{in}\ a\ b$$

puis, après conversion de clôture,

$$\lambda x.\ \lambda y.\ \mathbf{let}\ a \Leftarrow \lambda y.\ \lambda z.\ y\ \mathbf{in}\ \mathbf{let}\ b \Leftarrow \lambda x.\ \lambda y.\ \lambda z.\ y\ x\ \mathbf{in}\ a\ b$$

et enfin, après avoir hissé toutes les définitions à la racine,

$$\mathbf{let}\ a \Leftarrow \lambda y.\ \lambda z.\ y\ \mathbf{in}\ \mathbf{let}\ b \Leftarrow \lambda x.\ \lambda y.\ \lambda z.\ y\ x\ \mathbf{in}\ \lambda x.\ \lambda y.\ (a\ y)\ (b\ x\ y).$$

Après conversion de clôture, la taille de l'environnement pour chaque clôture manipulée par la machine de Krivine est minimale et manifeste. Un compilateur peut donc optimiser ces clôtures en employant des tableaux pour représenter tous les environnements.

Il existe d'autres variantes de la même transformation, en particulier la défonctionalisation (Reynolds, 1972), la transformation en supercombinateurs (Hughes, 1982) et le « lambda lifting » de Johnsson (1985). La transformation présentée ci-dessus a l'avantage d'opérer entièrement sur le λ-calcul pur avec définitions, de limiter le nombre d'arguments supplémentaires par rapport à d'autres transformations et d'être modulaire (il n'est pas nécessaire de connaître tout

le programme pour l'appliquer). Par simplicité, contrairement aux présentations usuelles de la conversion de clôture, nous représentons l'environnement à l'aide d'asbtractions supplémentaires, comme pour les transformations de Johnsson (1985) et Hughes (1982).

1.7 Conclusion

La présentation « papier » du λ-calcul pur est un cadre avantageux pour raisonner sur de nombreuses fonctions mathématiques en faisant l'économie de mécanismes ad hoc. Allié à des principes de raisonnement puissants tels que la convention de Barendregt pour le nommage de variables, il est possible de s'affranchir de quelques unes des difficultés techniques de ce cadre tel que le phénomène de capture de variables. Ces difficultés techniques ne peuvent être contournées ainsi quand il s'agit de calculer avec les termes du langage. Nous avons donc reformulé le λ-calcul pour aboutir, après de nombreuses étapes intermédiaires, à du code machine directement exécutable par un ordinateur. Nous résumons ici l'intérêt de chacune de ces étapes :

- les pointeurs à la place des noms permettent d'éviter de calculer l'ensemble des variables libres d'un terme avant chaque substitution ;
- passer à un langage avec substitutions explicites permet de modéliser le coût de la β-réduction et aussi de combiner les substitutions pour les appliquer en parallèle ;
- le choix d'une stratégie de réduction faible permet de ne pas propager les substitutions sous les abstractions et ainsi de ne jamais avoir besoin de renuméroter les pointeurs ballants d'un terme ;
- les machines abstraites simulent les systèmes de réécriture et leur recherche de radicaux sous une stratégie fixe aussi efficacement qu'une fonction d'évaluation mais tout en restant un modèle de calcul où la transition d'un état à un autre et le flot de contrôle est explicite ;
- la compilation des instructions d'une machine virtuelle en instructions natives de la machine réelle permet de simuler la machine virtuelle ailleurs que sur une feuille de papier et sans payer le coût de gestion d'autres méthodes de simulation telle que l'interprétation.

Notons que la machine de Krivine qui formait le fil conducteur des dernières sections est une machine abstraite parmi d'autres pour trouver la forme normale de tête faible d'un terme. Cette machine est aussi simple que sa stratégie de réduction est inefficace, provoquant la duplication inutile de nombreux radicaux. Il existe aujourd'hui une ménagerie de machines abstraites pour le λ-calcul pour toutes sortes de stratégies de réduction aussi bien que pour d'autres langages.

Citons notamment la machine STG (Peyton-Jones, 1992) au coeur du compilateur GHC[11] et d'autres compilateurs pour le langage de programmation Haskell, ainsi que la ZAM (Leroy, 1990 et Grégoire, 2003) pour OCaml[12]. Notons également l'existence de machines abstraites réduisant non plus faiblement mais fortement les termes, telles que les machines de Crégut (Crégut, 1990 et Crégut, 2007) et Curien (1993).

[11] `http://www.haskell.org/ghc`
[12] `http://caml.inria.fr`

Chapitre 2
Systèmes de types et algorithmes

Si le λ-calcul s'est révélé être un modèle particulièrement simple et élégant pour tous les calculs possibles, cette application n'était que fortuite. La publication de *Principia Mathematica* de Whitehead et Russell (1910) donna une base formelle à une large part des mathématiques de l'époque. Mais l'ouvrage laissait ouvertes de nombreuses questions clés, en particulier celui du sens de la substitution, traitée de manière informelle. De cette question émergea la logique combinatoire de Curry (1934) et un prédécesseur du λ-calcul de Church (1932). Church cherche à partir de 1928 à concevoir un système formel plus naturel d'usage que la théorie des types de Russell et la théorie des ensembles de Zermelo, basé sur la notion de fonction plutôt que sur celle d'ensemble. Mais Kleene et Rosser (1936) montrent que ce système est incohérent. Étant donné une fonction $k = \lambda x.\ \neg(x\ x)$, le paradoxe de Kleene-Rosser est encodé par l'application $k\ k$. La substitution de la variable x pour l'argument de k donne $\neg(k\ k)$. Le terme $k\ k$ est donc équivalent à sa négation, ce qui est un paradoxe. Church (1936) extrait la partie pertinente au calcul de ce système pour former le λ-calcul, mais revient à son objectif initial avec l'invention du λ-calcul simplement typé (Church, 1940). Comme dans la théorie de Russell, l'idée est de restreindre l'ensemble des termes à l'aide de types afin d'éliminer les paradoxes. Nous présentons dans ce chapitre diverses généralisations moins conservatrices que la théorie initiale de Church tout mais aussi cohérentes. Nous présenterons ceux-ci dans le cadre très général des *Pure Type Systems* de (Barendregt, 1991).

2.1 le λ-calcul simplement typé

Dans le λ-calcul simplement typé et dans tous les autres calculs de cette section, on distingue les termes dits « bien typés » de ceux qui ne le sont pas. Pour presque tous ces calculs, ce prédicat est décidable.

Définition 2.1 (Types simples) L'ensemble des types simples est donné inductivement comme suit :

- ι est un type simple ;
- si σ et τ sont des types simples alors $\sigma \to \tau$ est un type simple.

$$\text{(var)}\ \frac{}{\Gamma,x:\tau \vdash x:\tau} \qquad \text{(weak)}\ \frac{\Gamma \vdash x:\tau}{\Gamma,y:\sigma \vdash x:\tau}$$

$$\text{(abs)}\ \frac{\Gamma,x:\sigma \vdash M:\tau}{\Gamma \vdash \lambda x:\sigma.\, M:\sigma\to\tau} \qquad \text{(app)}\ \frac{\Gamma \vdash M:\sigma\to\tau \qquad \Gamma \vdash N:\sigma}{\Gamma \vdash M\ N:\tau}$$

Figure 2.1 Système de type pour le λ-calcul simplement typé. Par hypothèse, $x \neq y$ ci-dessus.

Définition 2.2 (λ^{st}-termes) Les termes du λ-calcul simplement typés sont donnés inductivement par la grammaire en forme Backus-Naur suivante :

$$\begin{array}{lcll} \mathsf{Var} & \ni & x,y,z & \\ \mathsf{Term} & \ni & M,N & ::= \quad x \mid \lambda x:\tau.\, M \mid M\ N \end{array}$$

L'ensemble des termes bien typés est lui aussi donné inductivement, par la figure 2.1. Il est caractérisé par un jugement de la forme $\Gamma \vdash M:\tau$, où Γ est une liste d'hypothèses de typage pour chacune des variables libres dans M. Remarquons que le terme k de Kleene n'est pas un terme du λ-calcul simplement typé : il n'existe pas de dérivation de typage pour $(x\ x)$ puisque le type de x doit être de la forme $\sigma\to\tau$ d'après la première prémisse de la règle (app), et x doit être de la forme σ d'après la deuxième prémisse de la règle (app). Le type de x est donc une solution de l'équation $\sigma = \sigma\to\tau$, mais aucune des solutions de cette équation n'est de taille finie et donc aucune n'est un type simple.

On peut envisager d'utiliser le λ^{st}-calcul dans un système formel de deux manières différentes. La première consiste à l'utiliser comme une théorie. Ce λ-calcul débarrassé de termes dont la réduction est paradoxale devient utile pour écrire les formules énonçant des propositions de *Principia Mathematica* et d'autres systèmes formels. En logique propositionnelle, la règle de substitution décrète qu'à partir d'une formule Q, il est permis de déduire la formule $[P_1/p_1,\ldots,P_n/p_n]Q$. L'attrait majeur du λ^{st}-calcul ici est qu'il explique précisément ce que veut dire la substitution sur les formules. Avec deux types de base ι,o (le « type des individus » et le « type des propositions »), on peut écrire les connecteurs logiques comme des constantes dont les types sont comme suit,

$$\dot\top, \dot\bot : o$$
$$\dot\neg : o\to o$$
$$\dot\wedge, \dot\vee : o\to o\to o$$

ainsi qu'une famille de quantificateurs pour chaque type τ

$$\dot{\forall}_\tau, \dot{\exists}_\tau : (\tau \to o) \to o.$$

En alternative, étant donné une famille infinie de constantes $(=)_\tau$, il est possible d'écrire tous les connecteurs logiques sous la forme de termes du λ^{st}-calcul, de sorte que toute formule sans axiome peut être écrite sous la forme d'un λ^{st}-terme avec $(=)$ comme seule constante.

Une fois ce langage des formules établi, il est nécessaire de choisir un système de déduction, tel que la déduction naturelle de Gentzen (1934) ou un système à la Hilbert (1927), qui justifiera la validité de certaines de ces formules. Une dérivation dans l'un de ces systèmes constitue une preuve qu'une formule donnée est vraie.

Mais la deuxième manière d'utiliser le λ^{st}-calcul est d'utiliser le λ^{st}-calcul lui-même comme langage de preuves, en remarquant d'une part que les types de ce langage sont en correspondance bijective avec les formules de la logique minimale (où $\Rightarrow$ est le seul connecteur logique). En oubliant les termes dans la figure 2.1, nous pouvons alors lire les règles de typage comme un système de déduction pour les formules qui correspondent aux types. D'autre part, les termes déterminent de manière unique (modulo affaiblissement du contexte par la règle (weak)) les dérivations de typage. Les termes du λ^{st}-calcul sont donc en bijection modulo affaiblissement avec les dérivations de typage. On peut donc prendre les termes eux mêmes comme des preuves des formules correspondant à leur type.

C'est l'isomorphisme de Curry-Howard.

2.2 Le $\lambda\Pi$-calcul

L'inconvénient de cette deuxième approche est que les types du λ^{st}-calcul sont en correspondance avec un fragment très faible de la logique, de sorte que les λ^{st}-termes ne peuvent justifier qu'une petite fraction de formules. Le $\lambda\Pi$-calcul est une extension du langage des types du λ-calcul simplement typé que l'on peut lire à travers la lentille de Curry-Howard comme l'ajout d'un opérateur de quantification universelle sur les termes dans les formules logiques. On passe ainsi de la logique minimale propositionnelle à la logique minimale des prédicats.

Définition 2.3 ($\lambda\Pi^-$-calcul) On distingue trois catégories syntaxiques : les *genres*, les *familles de types* et les *termes* :

$$\begin{array}{rcccl} \mathsf{Kind} & \ni & K & ::= & \mathbf{Type} \mid \Pi x : A.\ K \\ \mathsf{Fam} & \ni & A, B & ::= & x \mid \Pi x : A.\ B \mid A\ M \\ \mathsf{Term} & \ni & M, N & ::= & x \mid \lambda x : A.\ M \mid M\ N \end{array}$$

Il est usuel d'écrire $A \to B$ pour $\Pi x : A.\ B$ si $x \notin \mathcal{FV}(B)$.

Le $\lambda\Pi^{-}$-calcul[13] permet l'expression par l'isomorphisme de Curry-Howard de formules beaucoup plus élaborées que le λ^{st}-calcul, en particulier grâce à l'introduction de la notion de famille de type : un type indexé par un terme. Étant donné un encodage des entiers, nous pouvons écrire par exemple la proposition

$$\Pi n : \mathsf{nat}.\ \mathsf{even}\ n \to \mathsf{odd}\ n \to \dot{\bot},$$

correspondant à la formule

$$\forall n.\ \mathsf{even}(n) \Rightarrow \mathsf{odd}(n) \Rightarrow \bot.$$

Pour résumer, nous avons maintenant à disposition un langage permettant d'exprimer des formules formées de prédicats portant sur des termes que l'on peut quantifier.

Il peut être pratique de former certains types par abstraction. Il devient alors utile de calculer sur ces types, pour identifier ces abstractions appliquées à des termes aux types qu'elles produisent. Par exemple, nous pourrons écrire la fonction produisant le type correspondant à la conjonction n-aire $\top \wedge \ldots \wedge \top$

$$\mathsf{fun} \mathrel{\hat{=}} \lambda n : (o \to o) \to o \to o.\ n\ (\lambda P\ : o.\ \top \wedge P\,)\ \top$$

puis identifier $\mathsf{fun}\ (\lambda s : o \to o.\ \lambda o : o.\ s\ s\ o)$ et $\top \wedge \top \wedge \top$. Mais nous avons déjà une notion de calcul sur les termes. Si nous unifions la syntaxe des termes et la syntaxe des types, alors les types héritent directement des constructions de calcul déjà définies sur les termes.

Définition 2.4 ($\lambda\Pi$-calcul) La syntaxe des termes et des types du $\lambda\Pi$-calcul est donné comme suit :

$$\mathsf{Term} \quad \ni \quad M, N, A, B \quad ::= \quad \mathbf{Type} \mid \mathbf{Kind} \mid x \mid \lambda x : A.\ M \mid \Pi x : A.\ B \mid M\ N$$

Les règles de typage du $\lambda\Pi$-calcul sont données dans la figure 2.2. Nous avons une règle pour chaque construction syntaxique. Mais à la différence du λ^{st}-calcul, nous avons une règle supplémentaire, ne correspondant à aucune construction syntaxique, permettant de remplacer un type par une autre dans la dérivation de typage si les deux types sont *convertibles*, c'est à dire β-équivalents. Un jugement de typage n'a de sens que si le contexte de typage est bien formé, c'est à dire que les termes dans toutes les hypothèses sont tous des genres ou des familles de types et que aucunes des hypothèses ne sont mutuellement dépendantes.

[13] Ce calcul est similaire au *Canonical LF* de Harper et Licata (2007). Comme lui, il n'y a pas de calcul sur les types, mais les termes ne sont pas forcément en forme normale.

$$\frac{}{[]\ \mathsf{wf}} \qquad \frac{\Gamma \vdash A : s}{\Gamma,x : A\ \mathsf{wf}}\ \text{si } x \notin \Gamma$$

$$(sort)\ \frac{\Gamma\ \mathsf{wf}}{\Gamma \vdash \mathbf{Type} : \mathbf{Kind}} \qquad (var)\ \frac{\Gamma\ \mathsf{wf} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$(abs)\ \frac{\Gamma,x : A \vdash M\ : B \qquad \Gamma \vdash \Pi x : A.\ B : s}{\Gamma \vdash \lambda x : A.\ M\ : \Pi x : A.\ B}$$

$$(prod)\ \frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma,x : A \vdash B : s}{\Gamma \vdash \Pi x : A.\ B : s}$$

$$(app)\ \frac{\Gamma \vdash M\ : \Pi x : A.\ B \qquad \Gamma \vdash N\ : A}{\Gamma \vdash M\ N\ : \{N/x\}B}$$

$$(conv)\ \frac{\Gamma \vdash A : s \qquad \Gamma \vdash B : s \qquad \Gamma \vdash M\ : A}{\Gamma \vdash M\ : B}\ \text{si } A \equiv_\beta B$$

Figure 2.2 Jugements de typage pour le $\lambda\Pi$-calcul. Ici $s \in \{\mathbf{Type}, \mathbf{Kind}\}$.

2.3 Les systèmes de types purs (PTS)

Les types du $\lambda\Pi$-calcul correspondent aux formules de la logique des prédicats et les termes sont des preuves de ces formules. Mais la théorie initiale des types de Church, où les termes du λ-calcul simplement typé représentent les formules plutôt que les preuves, permet d'écrire des spécifications très concises en exploitant la possibilité de quantification sur les prédicats aussi bien que sur les termes, tel que peut l'offrir une logique d'ordre supérieur. On veut par exemple pouvoir écrire le principe de Leibniz pour deux termes a et b comme $\forall P\,.\ P\,(a) = P\,(b)$. Bien que cette formule soit syntaxiquement correcte en logique d'ordre supérieur, il est nécessaire de passer par un encodage puis d'établir des axiomes sur les constantes introduites pour cet encodage en logique des prédicats, faute de pouvoir quantifier sur P. Exploiter la correspondance de Curry-Howard a donc un prix.

Une solution consiste à étendre plus encore le système de type. Le système F de Girard (1971) permet par exemple des types en correspondance avec les formules de la logique de second ordre, en permettant une quantification sur les types. De manière générale, Barendregt dégage trois axes selon lesquels le système initial de types simples peut être étendu :

- permettre aux types de dépendre de termes (comme dans le $\lambda\Pi$-calcul) ;
- permettre aux termes de dépendre de types (comme dans le système F) ;
- permettre aux types de dépendre de types.

La combinaison de ces trois axes orthogonaux permet la formation de nouveaux systèmes de types, tel que le système Fω ou le $\lambda\Pi\omega$-calcul. On obtient ainsi huit systèmes que l'on peut

placer aux huit coins d'un cube : le λ-cube de Barendregt (1991). Ce dernier montre de plus comment décrire tous les systèmes du cube et bien d'autres à l'aide d'une unique syntaxe pour les termes de ces systèmes et d'un unique schéma de règles de typage.

Définition 2.5 (Les termes des systèmes de type pur) Soit $\langle \mathcal{S},\mathcal{A},\mathcal{R}\rangle$ une structure telle que $\mathcal{S}$ est un ensemble quelconque, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ et $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$. Les éléments de $\mathcal{S}$,$\mathcal{A}$ et $\mathcal{R}$ sont des *sortes*, des *axiomes* et des *règles*. Alors l'ensemble des termes d'un *système de types pur (PTS)* $\langle \mathcal{S},\mathcal{A},\mathcal{R}\rangle$ est donné par

$$\begin{array}{rcll} \mathcal{S} & \ni & s, s_n & \\ \mathsf{Term} & \ni & M, N, A, B & ::= \quad s \mid x \mid \lambda x : A.\ M \mid \Pi x : A.\ B \mid M\ N \end{array}$$

Les règles de typage d'un PTS $\langle \mathcal{S},\mathcal{A},\mathcal{R}\rangle$ sont celles du $\lambda\Pi$-calcul où la règle $(prod)$ est remplacée par la règle suivante :

$$(prod)\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma,x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.\ B : s_3}\ \text{si } \langle s_1,s_2,s_3\rangle \in \mathcal{R}$$

L'ensemble des sortes $\mathcal{S}$ est une généralisation de l'ensemble $\{\mathbf{Type}, \mathbf{Kind}\}$ du $\lambda\Pi$-calcul. L'ensemble $\mathcal{A}$ d'axiomes permet de typer les sortes et les règles déterminent le type des produits dépendants.

Définition 2.6 (Forme η-longue) La présence de type permet de diriger la notion d'η-équivalence. Pour toute expression bien typée M, on peut ainsi énoncer la règle suivante, dont l'application est limitée par le type de l'expression M :

$$M : \Pi x : A.\ B \longrightarrow_\eta \lambda x : A.\ M\ x$$

Un terme où la règle ci-dessus ne peut être appliquée sur aucun des sous-termes sans introduire un nouveau β-radical est dit *terme en forme η-longue*.

Définition 2.7 (PTS fonctionnel) Un PTS $\langle \mathcal{S},\mathcal{A},\mathcal{R}\rangle$ est *fonctionnel* si :

- pour tout s_1,s_2,s_2' si $(s_1,s_2) \in \mathcal{A}$ et $(s_1,s_2') \in \mathcal{A}$ alors $s_2 = s_2'$;
- pour tout s_1,s_2,s_3,s_3', si $(s_1,s_2,s_3) \in \mathcal{R}$ et $(s_1,s_2,s_3') \in \mathcal{R}$ alors $s_3 = s_3'$.

Pour tout PTS, contextes de typage Γ,Δ,Σ et termes M,N,A,B,C de ce PTS, nous avons les propriétés suivantes (Barendregt, 1992).

Lemme 2.8 (Affaiblissement, délayage)
Si $\Gamma \vdash M : A$ et $\Delta \supseteq \Gamma$ est bien formé alors $\Delta \vdash M : A$.

Lemme 2.9 (Renforcement)
Si $\Gamma,x:B,\Delta \vdash M : A$ *et* $x \notin \Delta \cup \mathcal{FV}(M) \cup \mathcal{FV}(A)$ *alors* $\Gamma,\Delta \vdash M : A$.

Lemme 2.10 (Permutation)
Si $\Gamma,x:B,y:C,\Delta \vdash M : A$ *et* $x \notin \mathcal{FV}(C)$ *alors* $\Gamma,y:C,x:B,\Delta \vdash M : A$.

Lemme 2.11 (Substitution)
Si $\Gamma,x:B,\Delta \vdash M : A$ *et* $\Gamma \vdash N : B$ *alors* $\Gamma,\{N/x\}\Delta \vdash \{N/x\}M : \{N/x\}A$.

Tous les systèmes du cube peuvent être décrits avec le même ensemble de sortes $\{\mathbf{Type},\mathbf{Kind}\}$ (abrégé en $\{*,\Box\}$) et un ensemble d'axiomes $\{(*,\Box)\}$. Il est possible de reformuler le λ^{st}-calcul comme un PTS où l'unique règle est $\{(*,*,*)\}$. Le $\lambda\Pi$-calcul a comme ensemble de règles $\{(*,*,*),(*,\Box,\Box)\}$. Le système F permet des types *polymorphes*. Il est donné par les règles $\{(*,*,*),(\Box,*,*)\}$. Le calcul des constructions de Coquand et Huet (1988) est le système donné par la conjonction des règles pour le $\lambda\Pi$-calcul, le système F et la règle $(\Box,\Box,\Box)$.

2.4 Le calcul des constructions inductives (CCI)

L'expressivité du calcul des constructions permet d'encoder de nombreux objets, tels que les entiers naturels, sans avoir recours à des axiomes. On peut y exprimer bon nombre de constructions inductives par une généralisation de l'encodage de Church. Cette approche pose deux problèmes : d'une part, les encodages ne sont pas très naturels à manipuler. D'autre part, avec ces encodages, il n'est pas possible de prouver dans le calcul des constructions sans axiomes quelques propriétés basiques comme $0 \neq 1$ et l'injectivité des constructeurs.

Coquand et Paulin (1990) proposent donc d'étendre le calcul des constructions avec un mécanisme de définitions de types inductifs qui génère automatiquement les schémas d'induction (récurseurs) attenants et étend la notion de convertibilité avec de nouvelles règles donnant un sens calculatoire à ces récurseurs. Geuvers (1994) généralise plus encore le calcul en permettant des définitions récursives, plus souples à l'usage que les récurseurs.

Définition 2.12 (Termes du calcul des constructions inductives (CCI)) La syntaxe des termes et des types du CCI est la suivante :

$$
\begin{array}{rcll}
\mathsf{Ind} & \ni & I, J & \qquad \mathsf{Constr} \quad \ni \quad C_n \\
\mathsf{Term} & \ni & M, N, A, B \quad ::= & x \mid c \mid C_i \mid I \mid \lambda x : A.\ M \mid \Pi x : A.\ B \mid M\ N \\
 & & & \mid \mathbf{let}\ x := N\ \mathbf{in}\ M \mid \mathbf{case}_I\ (M_S, M_P, N_1 | \ldots | N_n) \\
 & & & \mid \mathbf{fix}_k\ (f : A := M)
\end{array}
$$

Dans la construction d'analyse par cas, M_S est le terme à analyser, M_P est le prédicat donnant le type de retour et les termes $N_1, \ldots, N_n$ représentent chacune des branches. Le CCI inclut des opérateurs de points fixes permettant des définitions récursives, sous la condition d'un critère syntaxique de diminution de la taille du k-ième argument à chaque appel récursif.

Définition 2.13 (Type inductif) Dans le CCI, un *type inductif* de l *paramètres* et n *indices* est une constante de la forme

$$I : \Pi x_1 : A_1. \cdots \Pi x_l : A_l.\ \Pi x_{l+1} : A_{l+1}. \cdots \Pi x_n : A_n.\ s$$

auquel est associé zéro, un ou plusieurs constructeurs. Un *constructeur* C_i de I est une constante que l'on associe à I, dont le type est de la forme,

$$C_i : \Pi x_1 : A_1. \cdots \Pi x_l : A_l.\ \Pi y_1 : B_1. \cdots \Pi y_m : B_m.\ I\ A_1 \ldots A_l\ M_l \ldots M_n$$

Chaque constructeur d'un type inductif identifie une branche de toutes les analyses par cas de valeurs du type inductif I.

Définition 2.14 (Règles de réduction du CCI) La réduction en un pas des termes du CCI est donnée par les règles suivantes :

$$\Sigma \vdash (\lambda x.\ M)\ N \longrightarrow \{N/x\}M \qquad (\beta)$$

$$\Sigma \vdash c \longrightarrow M \qquad \text{si } (c := M : A) \in \Sigma \quad (\delta)$$

$$\Sigma \vdash \mathbf{let}\ x := N\ \mathbf{in}\ M \longrightarrow \{N/x\}M \qquad (\zeta)$$

$$\Sigma \vdash \mathbf{case}_I\ (C_i\ M_1 \ldots M_m, M_P, N_1 | \ldots | N_n) \longrightarrow N_i\ M_1 \ldots M_m \qquad (\iota)$$

$$\Sigma \vdash \mathbf{fix}_k\ (f : A := M)\ N_1 \ldots N_{k-1}\ (C_i\ L_1 \ldots L_n) \longrightarrow$$
$$\{\mathbf{fix}_k\ (f : A := M)/f\}M\ N_1 \ldots N_{k-1}\ (C_i\ L_1 \ldots L_n) \qquad (\textit{unfold})$$

Le CCI permet d'abréger la taille des termes en permettant le remplacement d'occurrences multiples de termes par un nom. La réduction est donc paramétrée par un environnement global Σ de définitions.

2.5 Types modulo

Dans le CCI tel que présenté ci-dessus, les récurseurs générés pour chaque type inductif voient leur comportement calculatoire spécifié en termes de points fixes. Une autre manière de donner le comportement calculatoire d'un récurseur rec_I est de donner des règles de réduction les concernant :

$$\mathsf{rec}_I\ f\ (C_i\ N_1\ \ldots\ N_n) \longrightarrow f\ N_1\ \ldots\ N_n\ (\mathsf{rec}_I\ f\ N_1)\ \ldots\ (\mathsf{rec}_I\ f\ N_n)$$

pour chaque constructeur C_i d'un type inductif I. C'est notamment le cas du système T de (Gödel, 1958). La déduction modulo (Dowek et al., 2003) choisit de placer points fixes avec analyses par cas, comme règles de réduction de récurseurs, dans le cadre plus général d'un calcul dont la règle de conversion est extensible avec des règles de réécriture arbitraires. On peut imaginer transposer cette idée à n'importe quel PTS. La règle $(conv)$ deviendrait alors

$$(conv)\ \frac{\Gamma \vdash A : s \qquad \Gamma \vdash B : s \qquad \Gamma \vdash M\ : A}{\Gamma \vdash M\ : B}\ \text{si}\ A \equiv_{\beta R} B$$

où $(\equiv_{\beta R})$ est la congruence générée par l'union de la règle (β) et d'un ensemble de règles de réécriture R arbitraire. Cousineau et Dowek (2007) montrent cependant que le $\lambda\Pi$-calcul avec règles de réécritures, baptisé $\lambda\Pi$-calcul modulo, est un cadre suffisant pour encoder tous les termes de tous les PTS fonctionnels — sans axiomes bien entendu.

Définition 2.15 (Plongement des PTS fonctionnels dans le $\lambda\Pi$-modulo) Soit un PTS $\mathcal{P} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$. Cousineau et Dowek (2007) construisent un contexte $\Sigma_{\mathcal{P}}$ contenant, pour chaque sorte $s \in \mathcal{S}$, deux hypothèses

$$U_s : \mathbf{Type} \text{ et } \epsilon_s : U_s \to \mathbf{Type},$$

pour chaque axiome $(s_1, s_2) \in \mathcal{A}$, une hypothèse

$$\dot{s_1} : U_{s_2},$$

et pour chaque règle $(s_1, s_2, s_3) \in \mathcal{R}$, une hypothèse

$$\dot{\Pi}_{\langle s_1, s_2, s_3 \rangle} : \Pi x : U_{s_1}.\ ((\epsilon_1\ x) \to U_{s_2}) \to U_{s_3}.$$

Le type U_s est l'*univers* de s et ϵ_s est la *fonction de décodage* de s. L'ensemble des règles de réécriture est formé de

$$\epsilon_{s_2}\ \dot{s_1} \longrightarrow U_{s_1}$$

pour chaque axiome dans $\mathcal{A}$ et de

$$\epsilon_{s_3}\ (\dot{\Pi}_{\langle s_1, s_2, s_3 \rangle}\ A\ B) \longrightarrow \Pi x : \epsilon_{s_1}\ A.\ \epsilon_{s_2}\ (B\ x)$$

pour chaque règle dans $\mathcal{R}$.
La traduction des termes de $\mathcal{P}$ vers le $\lambda\Pi$-calcul modulo est donnée par

- $|s| = \dot{s}$;
- $|x| = x$;

- $|\lambda x : A.\ M\,| = \lambda x : \epsilon_s\ |A|.\ |M\,|$;
- $|\Pi x : A.\ B| = \dot{\Pi}_{\langle s_1,s_2,s_3\rangle}\ |A|\ (\lambda x : \epsilon_{s_1}\ |A|.\ |B|)$, où s_1 est le type de A, s_2 est le type de B et s_3 est le type de $\Pi x : A.\ B$;
- $|M\ \ N\,| = |M\,|\ |N\,|$.

Nous montrerons dans le chapitre 5 comment étendre cette traduction au plongement du CCI.

2.6 Algorithmes de vérification

Un algorithme de vérification de types décide si une dérivation de typage est possible étant donné un contexte Γ, un terme et son type. Le plus souvent, on écrira un tel algorithme en terme d'une *inférence* de types, c'est à dire une fonction $\mathsf{Ctx} \times \mathsf{Term} \mapsto \mathsf{Type}$.

Un tel algorithme pour le λ-calcul simplement typé peut s'écrire comme une fonction (partielle) avec une clause pour chacune des règles de typage de ce système de types. La sélection de chacune des clauses est dirigée par la forme syntaxique du terme à typer :

$$\begin{aligned}
\mathsf{of}\ (\Gamma, y : A)\ x &= x && \text{si } x = y\\
\mathsf{of}\ (\Gamma, y : A)\ x &= \mathsf{of}\ \Gamma\ x && \text{sinon}\\
\mathsf{of}\ \Gamma\ (\lambda x : \sigma.\ M\,) &= \sigma \to (\mathsf{of}\ \Gamma\ M\,)\\
\mathsf{of}\ \Gamma\ (M\ \ N\,) &= \tau && \text{si } \sigma \to \tau = of\ \ \Gamma\ M \text{ et } of\ \ \Gamma\ N = \sigma
\end{aligned}$$

Mais remarquons que le $\lambda\Pi$-calcul et les PTS en général ne sont pas des systèmes tout à fait dirigés par la syntaxe : la règle $(conv)$ peut s'appliquer indépendemment de la forme syntaxique de M dans un jugement de la forme $\Gamma \vdash M : A$. En général, déterminer où placer les conversions pour reconstruire une dérivation complète de typage est indécidable. Si le PTS ne normalise pas, décider si deux types sont convertibles est déjà indécidable. van Benthem Jutting (1993) montre cependant que si le PTS est normalisant et fonctionnel, alors la vérification de types devient décidable. van Benthem Jutting et al. (1993) donnent une présentation des PTS dirigée par la syntaxe, où la conversion se fait à la volée dans chacune des règles de typage. Les règles de ce système sont donnée par la figure 2.3. Comme pour le λ^{st}-calcul, il est possible de les lire comme les clauses d'une fonction partielle d'inférence de types.

$$\frac{}{[]\ \mathsf{wf}} \qquad \frac{\Gamma \vdash A \twoheadrightarrow s}{\Gamma,x : A\ \mathsf{wf}}\ \text{si } x \notin \Gamma$$

$$(sort_{syn})\ \frac{\Gamma\ \mathsf{wf}}{\Gamma \vdash s_1 : s_2}\ \text{si } \langle s_1,s_2\rangle \in \mathcal{A} \qquad (var_{syn})\ \frac{\Gamma\ \mathsf{wf} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$(abs_{syn})\ \frac{\Gamma,x : A \vdash M \twoheadrightarrow B \qquad \Gamma \vdash \Pi x : A.\ B \twoheadrightarrow s}{\Gamma \vdash \lambda x : A.\ M \twoheadrightarrow \Pi x : A.\ B}$$

$$(prod_{syn})\ \frac{\Gamma \vdash A \twoheadrightarrow s_1 \qquad \Gamma,x : A \vdash B \twoheadrightarrow s_2}{\Gamma \vdash \Pi x : A.\ B \twoheadrightarrow s_3}\ \text{si } \langle s_1,s_2,s_3\rangle \in \mathcal{R}$$

$$(app_{syn})\ \frac{\Gamma \vdash M \twoheadrightarrow \Pi x : A.\ B \qquad \Gamma \vdash N \twoheadrightarrow A}{\Gamma \vdash M\ N : \{N/x\}B}$$

Figure 2.3 Système de type dirigé par la syntaxe pour un PTS donné par un tuple $\langle \mathcal{S},\mathcal{A},\mathcal{R}\rangle$ de sortes, d'axiomes et de règles. Nous écrivons $\Gamma \vdash M \twoheadrightarrow A$ si il existe A_0 tel que $\Gamma \vdash M : A_0$ et $A_0 \equiv_\beta A$.

chapitre 3
Conversion par évaluation

La montée en puissance des systèmes de types décrits dans le chapitre 2 est accompagnée d'une charge de plus en plus lourde pour le vérificateur de types. Là où tous les types simples sont des structures fondées[14], les types du système F peuvent voir leurs variables instanciées ou abstraites. Là où tous les types du système F se confondent à leur forme canonique, les types des PTS sont interchangeables avec d'autres membres d'une même classe d'équivalence de taille infinie. Même pour un PTS fortement normalisant, décider si deux termes appartiennent à la même classe d'équivalence générée par $(\equiv_\beta)$ est un problème arbitrairement difficile.

L'objectif de ce chapitre est double : celui d'arriver à une méthode *efficace* pour décider de la convertibilité de deux termes du $\lambda\Pi$-modulo, *à peu de frais* pour le programmeur de l'assistant de preuves.

3.1 Un algorithme de décision naïf

Deux termes M, N sont convertibles si $M \equiv_\beta N$. La relation $(\equiv_\beta)$ est une relation d'équivalence ; clôture réflexive, symétrique et transitive de la clôture par contexte de $(\longrightarrow_\beta)$. Quotienter l'ensemble des termes du λ-calcul par cette relation d'équivalence donne un ensemble $\Lambda/\equiv_\beta$ de classes d'équivalence. Tous les termes du λ-calcul sont membres d'exactement une classe d'équivalence tirée de $\Lambda/\equiv_\beta$. S'il est possible de choisir un représentant canonique pour chacune de ces classes d'équivalence, alors M, N sont convertibles si et seulement si M, N ont le même représentant canonique.

Bien qu'il existe de nombreux choix de représentations canoniques, si l'ensemble des termes considérés n'est plus tout le λ-calcul mais un ensemble de termes M tels que $\mathcal{SN}\ M$, alors les formes normales sont des représentants particulièrement naturels et pratiques de leurs classes d'équivalences respectives.

Étant donné un fonction de normalisation nf, un algorithme simple pour décider de la convertibilité de deux termes est :

$$\mathsf{conv}(N, M) = \mathsf{true} \quad \text{si } \mathsf{nf}(M) \equiv_\alpha \mathsf{nf}(N)$$
$$\mathsf{conv}(N, M) = \mathsf{false} \quad \text{sinon}$$

[14] « Ground structure » en anglais, c'est-à-dire sans occurrences de variables.

Cet algorithme est adéquat lorsque les formes normales des termes à vérifier ne sont pas grosses et sont rapides à calculer. C'est le cas de nombreux développements, tels que la métathéorie de langages logiques et de programmation (Harper et Licata, 2007). Certains assistants de preuve tels que TWELF et BELUGA (Pfenning et Schürmann, 1999 et Pientka et Dunfield, 2010) limitent même la grammaire des termes aux seules formes normales, stables par une notion de substitution adaptée appelée *substitution héréditaire*.

Mais le temps de vérification des termes typés d'un PTS peut vite dégénérer si les formes normales sont sensiblement plus grosses que les termes M ,N donnés en arguments à conv. D'autant plus que bien souvent, M et N sont syntaxiquement égaux modulo renommage éventuel de variables. Dans ce cas, M ,N appartiennent à la même classe d'équivalence par la propriété de Church-Rosser et il n'est nullement nécessaire de calculer les formes normales de M et N. Un algorithme un peu moins naïf consiste à n'appeler nf que si $M \not\equiv_\alpha N$:

$$\begin{aligned}
\mathsf{conv}(N\,,M\,) &= \mathsf{true} && \text{si } M \equiv_\alpha N \\
\mathsf{conv}(N\,,M\,) &= \mathsf{conv}'(N\,,M\,) && \text{sinon} \\
\mathsf{conv}'(N\,,M\,) &= \mathsf{true} && \text{si } \mathsf{nf}(M\,) \equiv_\alpha \mathsf{nf}(N\,) \\
\mathsf{conv}'(N\,,M\,) &= \mathsf{false} && \text{sinon}
\end{aligned}$$

Une stratégie moins naïve encore serait de tester l'égalité syntaxique après chaque étape de réduction, afin d'augmenter les opportunités de court-circuiter la normalisation et ainsi éviter plus encore des réductions futiles. Mais ces tests syntaxiques à répétition sont prohibitifs dans la pratique et leur degré de succès est très variable selon les termes et les stratégies de normalisation employées.

3.2 Normalisation par évaluation

Normalisation par réduction La clé de voûte de l'algorithme de décision pour la conversion décrit ci-dessus est la fonction nf. La méthode d'implémentation usuelle de cette fonction consiste à définir une fonction récursive sélectionnant les radicaux d'un terme sous une stratégie fixe et contractant ces radicaux. À la manière des optimisations successives décrites dans le chapitre 1 pour les évaluateurs, nous pourrions espérer obtenir un normaliseur plus efficace en employant des substitutions explicites pour appliquer les substitutions de manière paresseuse. Nous pourrions aller plus loin encore pour obtenir après transformations successives une machine abstraite pour la normalisation, et même compiler ces termes vers du code natif. C'est l'approche choisie par Crégut (1990,2007), qui propose la machine KN, variante de la machine

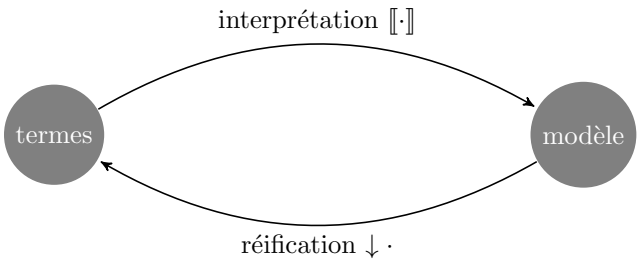


Figure 3.1 Normalisation par évaluation.

de Krivine produisant des formes normales et non des formes normales de tête faibles, à partir de laquelle Crégut écrit un compilateur vers du code natif pour processeur 68k.

Normalisation sans réduction L'approche précédente est profondément ancrée dans une vision de la normalisation comme un calcul qui est opéré étape par étape, du terme de départ vers la forme normale. Suivant Martin-Löf (1975a), une série d'auteurs (Berger et al., 1998, Coquand et Dybjer, 1997, Fiore et Simpson, 1999, Altenkirch et al., 1995 et Danvy, 2009) apportent cependant une notion plus abstraite de la normalisation. Ceux-ci considèrent une relation d'équivalence entre un terme et sa forme normale. Cette relation peut bien sûr être la clôture réflexive, symétrique et transitive d'une relation de réduction. Mais on peut tout aussi bien utiliser toute autre congruence sur les termes, prise comme une entité abstraite dont on ne sait si elle est définie en terme de réduction, un programme, une construction en théorie des ensembles ou encore par un oracle, entre autres congruences.

Une congruence particulièrement élégante est fondée sur l'idée de construction d'un modèle dénotationnel du λ-calcul à partir duquel il est possible d'extraire des formes syntaxiques résiduelles. Un tel modèle est dit *résidualisant*. Puisqu'un modèle dénotationnel du λ-calcul identifie les termes équivalents, postuler l'existence d'un modèle résidualisant permet de définir une fonction de normalisation comme la composition de la fonction d'interprétation vers le modèle $[\![\cdot]\!]$ avec la fonction d'extraction $(\downarrow)$ de la syntaxe résiduelle. La fonction $(\downarrow)$ est appelée *réification* (ou « quote », selon les auteurs). La fonction de normalisation construite à partir d'un modèle résidualisant est une fonction de *normalisation par évaluation (NbE)*, une instance de l'idée de normalisation sans réduction. Pour résumer, les deux propriétés essentielles ici sont que

1. $\forall M . \forall N . \; M \equiv_\beta N \Rightarrow [\![M]\!] \equiv_\alpha [\![N]\!]$ modulo renommage de variables (correction),
2. $\forall M . \; \downarrow [\![M]\!] = M$ si M est en forme normale (reproduction).

pour une notion d'équivalence $(\equiv)$ donnée ($\beta\eta$-équivalence, $\mathcal{R}$-équivalence pour un système de règles de réécriture $\mathcal{R}$, etc). Ces deux conditions sont suffisantes pour que $\downarrow [\![M]\!]$ donne la

$$
\begin{aligned}
\downarrow^{\iota} M &= M \\
\downarrow^{\sigma\to\tau} M &= \underline{\lambda} x : \sigma .\ \downarrow^{\tau} (M \mathbin{\bar{\cdot}} (\uparrow^{\sigma} x)) \quad \text{où } x \text{ est frais.} \\
\uparrow^{\iota} M &= M \\
\uparrow^{\sigma\to\tau} M &= \overline{\lambda} x : \sigma .\ \uparrow^{\tau} (M \mathbin{\underline{\cdot}} (\downarrow^{\sigma} x))
\end{aligned}
$$

Figure 3.2 Normalisation par évaluation pour le λ-calcul simplement typé. Nous distinguons abstractions et applications (que l'on écrit ici explicitement $\cdot$) du langage objet (souligné) de ceux du domaine d'interprétation (surligné).

forme normale d'un terme M quelconque si cette forme normale existe : nommons N cette forme normale et nous avons alors que $\downarrow [\![M]\!] = \downarrow [\![N]\!] = N$ par correction puis par reproduction.

Rien ne dit *a priori* que la fonction de normalisation définie de la sorte est effectivement calculable. Mais Martin-Löf (1975b) remarque que si la construction du modèle se fait dans une théorie intuitionniste, alors il est possible de donner une interprétation calculatoire à la fonction de normalisation ainsi obtenue, de sorte que la fonction de normalisation est véritablement un algorithme de normalisation. Mais à la différence d'une fonction de normalisation dont l'algorithme est l'itération d'une recherche de radicaux suivis d'une contraction du radical choisi, la correction de la fonction de normalisation obtenue par une telle construction sémantique est une conséquence de la construction du modèle, plutôt qu'une conséquence de propriétés de la théorie de la réécriture telle que la propriété de Church-Rosser.

Berger et al. (1998) proposent un algorithme particulièrement simple de normalisation par évaluation pour le λ-calcul simplement typé qui identifie tous les termes $\beta\eta$-équivalents avec leur forme β-normale et η-longue (voir définition 2.6). La fonction $(\downarrow)$ y est vue comme une fonction « inverse » à la fonction d'interprétation, définie par induction mutuelle sur les types du terme à réifier avec une fonction duale $(\uparrow)$ appelée fonction de réflexion (figure 3.2).

Le résultat de la réification est une forme normale par construction. Berger et Schwichtenberg démontrent que la forme normale $\downarrow^{\tau} [\![M]\!]$ est $\beta\eta$-équivalente à M. Berger et al. (2006) montrent par la suite que cet algorithme peut s'obtenir en appliquant la correspondance de Curry-Howard à une preuve de normalisation.

Cet algorithme est généralisé par la suite selon deux axes :

1. extension de l'algorithme vers des systèmes de types plus puissants, et
2. prise en compte de relations de conversion plus ou moins puissantes que $(\equiv_{\beta\eta})$.

Ainsi, plusieurs algorithmes de normalisation par évaluation sont proposés indépendemment pour le système F (Altenkirch et al., 2002, Vestergaard, 2001 et Abel, 2008), le système Fω (Abel, 2009) et diverses théories avec types dépendants, à la Martin-Löf (Abel et al., 2007a et Abel et al., 2007b) ou à la Coquand (Abel, 2010).

S'inspirant de travaux antérieurs sur l'évaluateur partiel Similix (Bondorf et Danvy, 1991) et la hiérarchie CPS (Danvy et Filinski, 1990), Danvy (1996) développe indépendemment le même algorithme que Berger et Schwichtenberg pour l'appliquer à l'évaluation partielle de programmes fonctionnels, puis le généralise au λ-calcul simplement typé avec produits et sommes, définitions récursives et aux fonctions avec effets de bords. La présence d'effets de bord et la possible divergence des termes à normaliser rend le résultat de l'algorithme sensible à la stratégie d'évaluation. Aussi, Danvy adapte celui-ci au λ-calcul en appel par valeur pour évaluer partiellement des programmes en ML.

Mais la relation d'équivalence qui sous-tend la TDPE de Danvy est plus faible que la $\beta\eta$-équivalence, dans la mesure où les formes canoniques obtenues par la TDPE de certains termes extensionnellement équivalents ne sont pas les mêmes. Définir une bonne notion de formes canoniques pour le λ-calcul extensionnel simplement typé avec sommes fortes est en fait étonnamment difficile et constitue un excellent exemple d'une application bénéfique de la NbE. Il est en effet difficile d'énoncer l'équivalence extensionnelle $(\equiv_{\beta\eta})$ pour un tel système comme un système de réécriture confluent et fortement normalisant. Un des rares algorithmes de décision basés sur la réécriture (Ghani, 1995) utilise un système de règles de réécriture non normalisant. Fiore et al. (2006) montrent par ailleurs qu'en présence de sommes fortes, la $\beta\eta$-équivalence n'admet pas d'axiomatisation finie, de manière similaire à la réponse par la négative de Wilkie (2000) et Gurevič (1985) au « problème des égalités du lycée » que Tarski posa en 1969 (Burris et Lee, 1993). Balat et al. (2004) caractérisent cependant des formes β-normales η-longues restreintes et démontrent qu'il est possible d'adapter la TDPE de Danvy pour produire ces formes restreintes, qui elles sont canoniques pour $(\equiv_{\beta\eta})$. Barral (2008) implémente un algorithme avec des exceptions comme seuls effets de contrôle, à la différence de la TDPE de Danvy (opérateurs statiques de contrôle délimité shift/reset (Danvy et Filinski, 1990)) et de l'algorithme de Balat et al. (opérateurs dynamiques de contrôle délimité set/cupto (Gunter et al., 1995)).

Berger et al. (1998,2003) étudient des extensions modulaires de la relation d'équivalence entre termes simplement typés par des règles de réécriture attachées à des constantes.

Normalisation sans réduction et sans types Normaliser sans réduction en exploitant le type des termes, tel que le font tous les algorithmes précédents, permet l'emploi de domaines sémantiques particulièrement simples. Les types permettent en outre de récupérer suffisamment d'information sur le terme pour traiter son interprétation dans le domaine sémantique comme

une boîte noire. Ces algorithmes sont donc très génériques en leur domaine sémantique — la seule opération requise sur une valeur sémantique est de pouvoir l'appliquer. Nous sommes donc libres de remplacer le λ-calcul par un domaine plus compliqué mais opérationnellement plus performant, tel qu'un calcul de clôtures, une machine abstraite ou même du code natif. Berger et al. (1998), Danvy et Vestergaard (1996), Danvy (1996) et Lindley (2005) rapportent ainsi des gains de performance de plusieurs ordres de grandeur avec la NbE et la TDPE par rapport à d'autres normaliseurs et d'autres technologies d'évaluation partielle.

Il est possible pourtant d'appliquer la même idée de normalisation sans réduction dans un contexte non typé. Dans ce cas, la relation d'équivalence considérée est la seule β-équivalence, puisqu'il est difficile d'offrir des formes canoniques pour la η-équivalence sans information de types pour guider les expansions. Filinski et Rohde (2004) proposent un algorithme de normalisation pour le λ-calcul pur où le domaine d'interprétation est un langage fonctionnel avec sommes. Filinski offre une analyse en théorie des domaines de la correction de l'algorithme, dont l'implémentation dans un langage de programmation fonctionnel quelconque peut être validée si le modèle est calculatoirement adéquat par rapport à la sémantique opérationnelle du langage. Aehlig et Joachimski (2004) arrivent à un algorithme similaire mais justifient la correction par des méthodes syntaxiques, en particulier des résultats de la théorie de la réécriture d'ordre supérieur, à l'inverse de la vision très sémantique des approches précédentes.

Nous proposons dans ce chapitre un algorithme de normalisation par évaluation pour le $\lambda\Pi$-modulo. Nous choisissons une approche non typée, car le $\lambda\Pi$-modulo est une théorie des types arbitrairement extensible par le biais de règles de réécriture définies par l'utilisateur. Nous perdons en cela l'élégante simplicité des modèles utilisés dans la normalisation par évaluation typée, ainsi que le traitement simple, uniforme et efficace de l'extensionnalité typiquement offerte par celle-ci. Pour autant, la congruence minimale sur les termes en $\lambda\Pi$-modulo est $(\equiv_\beta)$ et l'extensionnalité est absente de nombre de théories que nous souhaitons encoder dans $\lambda\Pi$-modulo. Celle-ci peut toujours être rajouté à l'aide d'un système de règles approprié, en fonction des besoins, dans le cas de théories extensionnelles.

En $\lambda\Pi$-modulo comme dans de nombreuses autres théories, les types ne jouent pas un rôle prépondérant dans la conversion ni la normalisation des termes. Une piste intéressante, que nous n'empruntons pas ici, serait d'utiliser les informations de typage pour décider plus rapidement de la convertibilité de deux termes, à l'aide de « théorèmes gratuits » par exemple (Wadler, 1989). Mais en général, ces théorèmes sont trop faible pour être utiles. Nous observons aussi que les types jouent un rôle marginal, voir inexistant, durant l'optimisation de programmes fonctionnels par la plupart des compilateurs. Nous ne perdons donc pas beaucoup d'opportunités d'optimisation en oubliant le type des termes pour tester la conversion, *a priori*.

L'approche que nous adoptons est dans la droite ligne de travaux antérieurs de Grégoire et Leroy (2002) sur l'implémentation du test de conversion par compilation des termes. Comme le leur, l'algorithme présenté dans ce chapitre réutilise des implémentations existantes de compilateurs pour des variantes du λ-calcul. Comme le leur, l'algorithme est dirigée par la syntaxe des formes normales de tête faible et non par les types. L'algorithme de Grégoire et Leroy procède par interprétation des termes à tester vers des formes normales de têtes faible, et s'apparente ainsi à de la NbE. Leur phase dite de *relecture* de formes neutres est très similaire à l'opération de réification de la NbE non typée de Filinski.

À la différence de l'algorithme de Grégoire et Leroy, notre approche est entièrement portable et ne demande aucune modification de l'évaluateur sous-jacent. L'implémentation de leur « calcul symbolique » dans Coq est une modification de la machine virtuelle standard de OCaml, la ZAM.

Aehlig et al. (2008) implémentent aussi un normaliseur non typé pour des termes typés, dans le cadre du simplifieur de Isabelle/HOL. Leur algorithme, une variante de NbE pour un système de règles de réécriture arbitraire, ne demande aucune modification de l'évaluateur sous-jacent, ce qui leur permet de réutiliser à peu de frais un compilateur pour Standard ML. Mais leur encodage des règles de réécriture souffre de nombreuses allocations et déallocations de structures de listes ainsi que de clôtures intermédiaires avec des durées de vie très courtes, de sorte que la performance de leur normalisation de termes compilés vers du code natif est plus lente que la machine virtuelle de Coq, qui pourtant manipule du bytecode. Nous montrons comment implémenter la NbE en présence de règles de réécriture sans allocation de listes et en réduisant grandement le nombre de clôtures allouées durant la normalisation.

3.3 La normalisation par évaluation est un auto-réduiseur

La machine de Turing universelle simule l'exécution de toutes les autres machines de Turing, même une autre machine de Turing universelle. Autrement dit, la machine de Turing universelle est un interpréteur de machines de Turing. Un interpréteur capable de s'interpréter soi-même est un *auto-interpréteur*. Étant donné un opérateur injectif $\ulcorner\cdot\urcorner$ (le *schéma de représentation*), Mogensen (1992) transpose la notion d'auto-interpréteur au λ-calcul pur ainsi :

$$E \ulcorner M \urcorner \equiv_\beta M\,.$$

Un auto-interpréteur est un terme E tel que pour tout terme M, l'application de E à la représentation $\ulcorner M \urcorner$ est convertible à M. Mogensen définit également une notion associée *d'auto-réduiseur* R tel que :

$$R \ulcorner M \urcorner \equiv_\beta \ulcorner M' \urcorner$$

où M' est la forme normale de M, si et seulement si cette forme normale existe. La différence essentielle entre un auto-interpréteur et un auto-réduiseur est que l'application de ce dernier à un terme se réduit en la représentation de la forme normale de son argument, alors que l'application de l'auto-interpréteur se réduit en la forme normale de son argument.

Dans ce qui suit, nous soulignerons les objets syntaxiques et noterons l'application explicitement par $M \mathbin{\underline{\cdot}} N$, pour distinguer la syntaxe de l'interprétation.

L'opérateur $\ulcorner\cdot\urcorner$[15] ne peut pas bien sûr être défini dans le λ-calcul lui-même, mais Mogensen postule l'existence d'une telle primitive. La représentation d'un terme est une donnée, que l'on peut manipuler, transformer et inspecter dans le calcul lui-même. Il est naturel de représenter les données comme des termes en forme normale, de sorte que les données soient constantes par réduction. Le schéma de représentation est donc de type $\ulcorner\cdot\urcorner : \mathsf{Term} \to \mathsf{Term}^{\mathsf{NF}}$. Nous ajoutons qu'une bonne représentation est aussi un terme « paramétrique », au sens où sa valeur ne dépend pas de la valeur des variables dans la représentation.

Définition 3.1 (Terme paramétrique) Un terme en forme normale M est *paramétrique* si pour toutes les variables x dans M, le remplacement de x par tout terme en forme normale N dans M, noté $M[N]$, est en forme normale.

Toutes sortes de représentations sont possibles, mais Mogensen présente une représentation des termes admettant un auto-interpréteur trivial. Soit B,F deux constructeurs unaires pour les variables liées et libres (respectivement), et Lam, App deux constructeurs binaires. Ceux-ci sont des termes du λ-calcul pur par une généralisation de l'encodage de Church aux types algébriques : l'encodage de Böhm et Berarducci (1985). Alors une représentation d'un terme (clos) du λ-calcul dans le λ-calcul est donnée par (Mogensen, 1992) :

$$\ulcorner x \urcorner = \mathsf{B}\ x$$
$$\ulcorner \underline{\lambda} x.\ M \urcorner = \mathsf{Lam}\ (\lambda x. \ulcorner M \urcorner)$$
$$\ulcorner M \mathbin{\underline{\cdot}} N \urcorner = \mathsf{App}\ \ulcorner M \urcorner \ulcorner N \urcorner$$

Ce schéma de représentation est un exemple de syntaxe abstraite d'ordre supérieur (HOAS), souvent attribuée à Pfenning et Elliott (1988) bien qu'apparaissant déjà dans (Reynolds, 1985). Mogensen donne un auto-interpréteur pour ce schéma de représentation,

[15] Une construction analogue à la forme `quote` de Scheme.

$$\begin{aligned}
\mathsf{eval}\ (\mathsf{B}\ x) &= x \\
\mathsf{eval}\ (\mathsf{Lam}\ f) &= \lambda x.\ \mathsf{eval}\ (f\ x) \\
\mathsf{eval}\ (\mathsf{App}\ M\ N) &= \mathsf{app}\ (\mathsf{eval}\ M)\ (\mathsf{eval}\ N) \\
\mathsf{app}\ f\ N &= f\,N
\end{aligned}$$

que nous reconnaissons comme le premier interpréteur définitionnel de (Reynolds, 1972) adapté à une syntaxe abstraite d'ordre supérieur. Comme le remarque Reynolds, cet interpréteur emprunte tellement du métalangage que même l'ordre d'évaluation du langage de cet interpréteur est fonction de l'ordre d'interprétation du métalangage.

Dans le vocabulaire de la normalisation par évaluation, le modèle est le λ-calcul lui-même et nous donnons l'interprétation des termes comme la composition du schéma de représentation et de l'évaluation d'un terme,

$$[\![M]\!] = \mathsf{eval}\ \ulcorner M \urcorner.$$

Mais il nous faut aussi pouvoir extraire de la syntaxe du modèle. Nous modifions donc l'évaluation pour que la réification des termes puisse être dirigée par la forme normale de son interprétation[16] :

$$\begin{aligned}
\mathsf{eval}\ (\mathsf{B}\ x) &= x \\
\mathsf{eval}\ (\mathsf{Lam}\ f) &= \mathsf{Lam}\ (\lambda x.\ \mathsf{eval}\ (f\ x)) \\
\mathsf{eval}\ (\mathsf{App}\ M\ N) &= \mathsf{app}\ (\mathsf{eval}\ M)\ (\mathsf{eval}\ N) \\
\mathsf{app}\ (\mathsf{Lam}\ f)\ N &= f\ N \\
\mathsf{app}\ M\ N &= \mathsf{App}\ M\ N
\end{aligned} \tag{3.1}$$

La fonction eval envoie des représentations de termes vers des représentations. Ici, l'interprétation de l'application est une fonction qui décapsule l'interprétation du membre gauche pour récupérer une fonction. Si cette décapsulation n'est pas possible, alors l'interprétation de l'application est une représentation de cette application[17]. Remarquons que l'évaluation est (presque) définie par induction sur les termes, comme l'est le schéma de représentation. Notons aussi que le schéma de représentation respecte aussi une propriété de bonne formation des représentations : que toutes les occurrences de variables du métalangage n'apparaissent que comme

[16] Les équations pour app se chevauchent ; on ne considère la deuxième que si la première ne s'applique pas.

[17] La représentation de l'application obtenue est un terme neutre, analogue aux accumulateurs du calcul symbolique de Grégoire et Leroy (2002).

arguments au constructeur B. Cette propriété de bonne formation des représentations garantit la paramétricité et nous permet ainsi de fusionner l'évaluation et le schéma de représentation :

$$\begin{aligned} [\![x]\!] &= x \\ [\![\underline{\lambda}x.\ M]\!] &= \mathsf{Lam}\ (\lambda x.\ [\![M]\!]) \\ [\![M\ \underline{\cdot}\ N]\!] &= \mathsf{app}\ [\![M]\!]\ [\![N]\!] \end{aligned} \tag{3.2}$$

La réification de l'interprétation d'un terme M donne la syntaxe correspondant à l'interprétation :

$$\begin{aligned} \downarrow_n \mathsf{F}\ m &= m \\ \downarrow_n \mathsf{Lam}\ f &= \underline{\lambda}n.\ \downarrow_{n+1} (f\ (\mathsf{F}\ n)) \\ \downarrow_n \mathsf{App}\ M\ N &= (\downarrow_n M)\underline{\cdot}(\downarrow_n N) \end{aligned}$$

La réification d'une application du modèle est une application syntaxique. La réification d'une abstraction du modèle est aussi une abstraction syntaxique. Le corps de celle-ci est la réification du corps de l'abstraction dans le modèle, où la variable liée par l'abstraction devient libre. L'opération de réification est paramétrée par un entier n jouant le rôle de compteur pour générer des noms de variables frais pour les variables libres[18]. Cet algorithme est similaire à la représentation « *locally nameless* » du λ-calcul (McKinna et Pollack, 1993, McBride et McKinna, 2004 et Aydemir et al., 2008), où seules les variables libres sont nommées, alors que les variables liées restent abstraites (dans les deux sens du mot). Naviguer sous un lieur libère une variable dont il faut alors choisir un nom.

Nous avons maintenant tous les ingrédients d'un normaliseur pour les termes clos du λ-calcul pur :

$$\mathsf{nf}\ M\ =\ \downarrow_0 \mathsf{eval}\ \ulcorner M \urcorner =\ \downarrow_0 [\![M]\!]$$

En réinjectant la forme normale obtenue par réification dans la modèle, on obtient un auto-réduiseur, c'est-à-dire une fonction de représentation à représentation. Cet auto-réduiseur est plus efficace que l'auto-réduiseur originel de Mogensen (1992) qui construit la forme normale à l'aide de représentations en paires avec leur sémantique[19], essentiellement contrainte par le fait que l'interprétation de Mogensen ne distingue par les variables libres des variables liées (à l'inverse de nos B et F).

Exemple 3.2 Considérons la normalisation du terme

[18] Ces noms peuvent être vus comme des niveaux de de Bruijn.
[19] Une construction similaire à la construction catégorique dite de « glueing » (Lafont, 1988appendice A).

$$M = (\underline{\lambda}x.\ \underline{\lambda}y.\ (\underline{\lambda}z.\ y \mathbin{\underline{\cdot}} x) \mathbin{\underline{\cdot}} x) \mathbin{\underline{\cdot}} (\underline{\lambda}x.\ x).$$

L'interprétation $[\![M]\!]$ de ce terme est

$$\mathsf{app}\ (\mathsf{Lam}\ (\lambda x.\ \mathsf{Lam}\ (\lambda y.\ \mathsf{app}\ (\mathsf{Lam}\ (\lambda z.\ \mathsf{app}\ y\ x))\ x)))\ (\mathsf{Lam}\ (\lambda x.\ x)).$$

La réification de l'interprétation du terme procède par étapes :

$$\begin{aligned}
\downarrow_0 M &\longrightarrow \downarrow_0 \mathsf{Lam}\ (\lambda y.\ \mathsf{app}\ (\mathsf{Lam}\ (\lambda z.\ \mathsf{app}\ y\ (\mathsf{Lam}\ (\lambda x.\ x))))\ (\mathsf{Lam}\ (\lambda x.\ x)))\\
&\longrightarrow \underline{\lambda}0.\ \downarrow_1 \mathsf{app}\ (\mathsf{Lam}\ (\lambda z.\ \mathsf{app}\ (\mathsf{F}\ 0)\ (\mathsf{Lam}\ (\lambda x.\ x))))\ (\mathsf{Lam}\ (\lambda x.\ x))\\
&\longrightarrow \underline{\lambda}0.\ \downarrow_1 \mathsf{app}\ (\mathsf{F}\ 0)\ (\mathsf{Lam}\ (\lambda x.\ x))\\
&\longrightarrow \underline{\lambda}0.\ \downarrow_1 \mathsf{App}\ (\mathsf{F}\ 0)\ (\mathsf{Lam}\ (\lambda x.\ x))\\
&\longrightarrow \underline{\lambda}0.\ (\downarrow_1 \mathsf{F}\ 0) \mathbin{\underline{\cdot}} (\downarrow_1 \mathsf{Lam}\ (\lambda x.\ x))\\
&\longrightarrow \underline{\lambda}0.\ 0 \mathbin{\underline{\cdot}} (\underline{\lambda}1.\ \downarrow_2 \mathsf{F}\ 1)\\
&\longrightarrow \underline{\lambda}0.\ 0 \mathbin{\underline{\cdot}} (\underline{\lambda}1.\ 1)
\end{aligned}$$

3.4 Normalisation multi-étage

Nous remarquons une occurrence sous un lieur du schéma de représentation dans sa définition. De même, les fonctions eval et $[\![\cdot]\!]$ apparaissent en position négative dans l'équation (3.1) et l'équation (3.2), respectivement. Rappelons que les stratégies usuelles d'évaluation des langages fonctionnels, en appel par nom ou en appel par valeur, ne réduisent jamais sous les abstractions (voir l'étape 3 du chapitre 1). Le radical (souligné) dans le terme $\lambda x.\ \underline{(\lambda y.\ y)\ x}$ ne sera donc réduit qu'après application de ce terme à un argument. Si le terme est appliqué plusieurs fois, alors le radical sera réduit plusieurs fois. Si la fonction d'interprétation des termes est implémentée comme une fonction dans un langage fonctionnel, alors pour un M fixé, les occurrences négatives de $\ulcorner\cdot\urcorner$ dans l'interprétation sont autant de radicaux qui ne seront réduits qu'après application de l'abstraction sous laquelle elles apparaissent. Dans l'exemple 3.2, l'interprétation $\mathsf{Lam}\ (\lambda x.\ x)$ est dupliqué et appliquée deux fois durant la réification. Ce terme est la forme normale de l'interprétation. Si la fonction d'interprétation est une fonction du langage de l'interprétation, la forme normale de tête faible est $\mathsf{Lam}\ (\lambda x.\ [\![x]\!])$. Elle contient un radical, qui sera réduit deux fois, après les deux applications.

Dans une implémentation concrète, la fonction d'interprétation doit donc être implémentée non pas comme une fonction du métalangage, mais comme une méta-fonction du métalangage, c'est-à-dire une fonction qui produit une représentation textuelle du terme. Il existe de nombreuses extensions de langages de programmation usuels permettant d'exprimer ce genre de

méta-calcul, ou programmation « multi-étage » (Sheard, 1999, Sheard et Peyton-Jones, 2002 et Taha, 2004). Cette programmation multi-étage peut aussi être simulée par un programme générateur de programmes fonctionnels, ou préprocesseur.

3.5 Règles de réécriture

Il est possible en $\lambda\Pi$-modulo de donner un contenu calculatoire à une constante en lui associant un ensemble de règles de réécriture. La convertibilité des termes est considérée modulo l'ensemble de toutes les règles de réécriture pour toutes les constantes apparaissant dans les termes. Soit $\mathcal{R}$ cet ensemble de règles. Un algorithme pour décider si deux termes M, N sont convertibles consiste à injecter M, N dans le λ-calcul non typé, puis comparer les formes normales pour la congruence $\equiv_{\beta\mathcal{R}}$ engendrée par l'union des règles (β) et de $\mathcal{R}$. Dans cette section, nous étendons l'algorithme de normalisation de la section précédente au λ-calcul non typé avec constantes et règles de réécriture sur certaines de ces constantes.

Cet algorithme de décision n'est complet que si l'ensemble $\mathcal{R}$ de règles de réécriture se comporte bien. En particulier, la confluence et la terminaison du système étendu avec (β) sont des propriétés fort désirables — faute de quoi les formes normales de M, N n'existent pas forcément et ne sont pas toujours canoniques si elles existent. En revanche, La correction de l'algorithme ne dépend d'aucune condition particulière sur les règles.

Nous choisissons un langage fonctionnel en appel par valeur ou en appel par nom comme modèle des termes à normaliser. Ce choix dicte également certaines restrictions draconiennes sur la stratégie de réduction des règles de réécriture. Nous considérons en particulier l'ensemble des règles données comme un ensemble ordonné. Pour une constante c d'arité n, une occurrence de c appliquée à n arguments dans le terme à normaliser est d'abord filtrée par le côté gauche de la première règle de réécriture associée à c. Si cette règle ne s'applique pas, alors on tente d'appliquer la deuxième règle, et ainsi de suite. Ainsi, si deux règles sont applicables, on choisit d'appliquer la première sans jamais revenir sur ce choix dans la suite de la réduction du terme vers sa forme normale. Cette stratégie de réduction est cohérente avec les constructions de filtrage par motif (constructions `case ... of` ou `match ... with`) que l'on trouve dans de nombreux langages de programmation. Qui plus est, la stratégie de sélection des occurrences de c à réduire est dictée par la sémantique opérationnelle du langage fonctionnel hôte.

Ce choix est dicté par deux impératifs : simplicité et performance. Les encodages des logiques fondatrices de la plupart des assistants de preuves, tels que le calcul des constructions inductives de Coq ou HOL de Isabelle, requièrent l'ajout de règles de réécriture relativement simples puisque ces logiques ne sont pas fondamentalement différentes du $\lambda\Pi$-modulo. Dans cette section, nous faisons le choix de privilégier l'interprétation simple et efficace de la β-réduction.

$$[\![\_]\!] = \_ \qquad [\![x]\!] = x$$
$$[\![c\ P_1\ \ldots\ P_n]\!] = \mathsf{App}\ (\ldots(\mathsf{App}\ (\mathsf{Con}\ \hat{c})\ [\![P_1]\!])\ldots)\ [\![P_n]\!]$$

Figure 3.3 Traduction des motifs de filtrage.

$$\left[\!\!\left[\begin{array}{lcl} c\ P_{11}\ \ldots\ P_{1n} & \longrightarrow & M_1 \\ \vdots & & \vdots \\ c\ P_{m1}\ \ldots\ P_{mn} & \longrightarrow & M_m \end{array}\right]\!\!\right] = \begin{array}{l} \mathbf{fix}\ (\lambda c.\ \lambda x_1.\ \cdots \lambda x_n. \\ \quad \mathbf{case}\ (x_1, \ldots, x_n)\ \mathbf{of} \\ \qquad \begin{array}{lcl} ([\![P_{11}]\!], \ldots, [\![P_{1n}]\!]) & \to & [\![M_1]\!] \\ \vdots & & \vdots \\ ([\![P_{m1}]\!], \ldots, [\![P_{mn}]\!]) & \to & [\![M_m]\!] \\ \mathbf{default} & \to & [\![c\ x_1\ \ldots\ x_n]\!] \end{array} \end{array}$$

Figure 3.4 Traduction des règles de réécriture pour une constante c d'arité n. $x_1, \ldots, x_n$ sont choisis tels que ceux-ci sont frais pour $M_1, \ldots, M_m$.

Nous souhaitons en particulier éviter les interprétations vers des langages de premier ordre qui demanderait de simuler les abstractions du λ-calcul par un encodage à base de substitutions explicites ou tout autre calcul de clôtures. Les langages fonctionnels avec filtrage par motifs se prêtent bien à l'exécution de systèmes de règles simples. Dans le cas de théories de graphes, associatives, ou associatives-commutatives, nous pourrions tout aussi bien viser des langages aux stratégies d'évaluation plus souples et aux mécanismes de réduction plus spécialisés et optimisés tels que ASF+SDF (van den Brand et al., 2001), Maude (Clavel et al., 2003) ou Tom (Balland et al., 2007).

La figure 3.4 étend l'interprétation de la section précédente aux ensembles de règles pour une constante c. Nous supposons ici que toutes les règles pour une même constante sont de même arité. Un ensemble de règles est interprété comme une fonction du métalangage. Comme la réduction d'une occurrence d'une constante peut créer un nouveau radical potentiel avec c en tête de ce radical, la fonction sémantique est définie récursivement à l'aide de l'opérateur de points fixes du métalangage.

L'ensemble des étiquettes $\mathsf{App}, \mathsf{F}, \mathsf{Lam}$ utilisées dans l'interprétation est étendu avec un nouveau constructeur unaire Con dont le premier argument contient une représentation du nom de la constante c. Si aucune des règles de réécriture n'est applicable à l'occurrence de c sélectionnée, ce qui est typiquement le cas lorsque c est appliquée à une variable libre, alors l'application de c à ses arguments n'est pas un radical. L'application de $c\ x_1\ \ldots\ x_n$ à des

arguments supplémentaires n'en sera pas plus. La branche par défaut du filtre construit donc une représentation neutre de l'application de la constante à ses arguments. Nous notons $\hat{c}$ la chaîne de caractères dans le métalangage construite à partir du nom c. La réification de la représentation d'une constante reconstitue la constante au niveau syntaxique à partir de son nom :

$$\downarrow_n \mathsf{Con}\ \hat{c} = c$$

Règles de réécriture non-linéaires Une règle de réécriture non-linéaire est une règle dont le côté gauche mentionne la même variable plus d'une fois. Il peut être pratique[20] d'employer ce genre de règles de réécriture, notamment dans l'encodage de récurseurs de théories inductives. Nous traitons ces règles comme du sucre syntaxique pour des règles conditionnelles, où les seules conditions possibles sont des contraintes de convertibilité sur certaines des variables apparaissant dans le membre gauche de la règle de réécriture. Ainsi, la règle de réécriture

$$\mathsf{eq}\ x\ x \longrightarrow \top$$

se traduit comme la branche suivante d'une construction de filtrage du métalangage :

$$(x_1,x_2)\ \mathbf{when}\ \downarrow_0 x_1 = \downarrow_0 x_2 \rightarrow [\![\top]\!]$$

Les règles non-linéaires sont linéarisées à gauche et les contraintes de convertibilité entre variables sont accumulées comme autant de conditions sur l'application de la règle.

3.6 Optimisations

3.6.1 Décurryfication

Dans la section 3.3, l'interprétation des termes que nous avons construit pour permettre la réification est fort coûteuse d'un point de vue opérationnel. Comme nous normalisons des termes non typés, il est nécessaire d'ajouter des étiquettes (dont les constructeurs F, Lam et App jouent le rôle) à chaque forme syntaxique dans l'interprétation. Si les termes de l'interprétation sont des programmes fonctionnels exécutés par compilation vers du code natif, ces étiquettes prennent de la place en mémoire durant l'exécution du code et ne permettent pas au compilateur d'éviter l'allocation de certaines clôtures à la durée de vie très courte.

Prenons les fonctions suivantes :

[20] Il faut être prudent : ajouter des règles de réécriture non linéaires est très dangereux pour la confluence du système. Klop (1980) montre un système de réécriture qui n'est pas confluent en dépit de l'absence de toute paire critique et de la confluence locale du système.

$$\begin{aligned}
\mathsf{nil} &\mathrel{\hat{=}} \lambda f\, g.\ f \\
\mathsf{cons} &\mathrel{\hat{=}} \lambda h t f\, g.\ g\ h\ (t\ f\ \ g) \\
\mathsf{map} &\mathrel{\hat{=}} \lambda f\, l.\ l\ \mathsf{nil}\ (\lambda h t.\ \mathsf{cons}\ (f\ \ h)\ t)
\end{aligned}$$

Les fonctions nil et cons sont les encodages de Böhm et Berarducci (1985) des constructeurs [] et (: :) pour les listes. La fonction map applique la fonction f à chaque élément de la liste donnée en argument. La notation $\lambda x_1 \cdots x_n.\ M$ est du sucre syntaxique pour $\lambda x_1.\ \cdots \lambda x_n.\ M$. Les fonctions ci-dessus sont écrites comme des fonctions d'arité supérieure, mais sont encodées par des fonctions unaires qui rendent d'autres fonctions en argument. Cet encodage est une *curryfication*.

Mais la curryfication a un coût. Appliquer une fonction curryfiée à une suite d'arguments signifie la création de nombreuses clôtures intermédiaires — une clôture pour chaque fonction obtenue par application d'une fonction à un argument. Il faudra allouer (puis désallouer plus tard) $n - 1$ clôtures pour l'application de n arguments. Dans la suite de réductions suivante,

$$\begin{aligned}
&[\![\mathsf{map}\ (\lambda x.\ x)\ \mathsf{nil}]\!] \\
&\quad = \quad \mathsf{app}\ (\mathsf{app}\ \mathsf{map}\ (\mathsf{Lam}\ (\lambda x.\ x)))\ \mathsf{nil} \\
&\quad = \quad \mathsf{app}\ (\mathsf{app}\ (\mathsf{Lam}\ (\lambda f\,.\ \mathsf{Lam}(\lambda l.\ \mathsf{app}\ (\mathsf{app}\ l\ \mathsf{nil})\ \ldots))))\ (\mathsf{Lam}\ (\lambda x.\ x)))\ \mathsf{nil} \\
&\quad \longrightarrow \quad \mathsf{app}\ ([\mathsf{Lam}\ (\lambda x.\ x)/f](\mathsf{Lam}\ \lambda l.\ \mathsf{app}\ (\mathsf{app}\ l\ \mathsf{nil})\ \ldots))\ \mathsf{nil} \\
&\quad \longrightarrow^* \quad \mathsf{nil}
\end{aligned}$$

la fonction map est appliquée à deux arguments. Une représentation Lam $(\lambda x.\ \ldots)$ intermédiaire est donc allouée, bien qu'un encodage alternatif des fonctions n-aires éviterait cela.

On retrouve dans la littérature de nombreux encodages de fonctions n-aires. Marlow et Peyton-Jones (2006) proposent la dichotomie Push/Enter et Eval/Apply pour les classifier. Dans le modèle Push/Enter, le bloc de code de la fonction appelante pousse tous les arguments disponibles sur une pile puis fait un saut vers (*entre*) dans le code de la fonction appelée. Celui-ci vérifie qu'il y a au moins autant d'arguments sur la pile que l'arité de la fonction, puis continue l'exécution. Sinon, le code appelé construit une clôture qui est retournée de suite au code appelant. Nous choisissons plutôt le modèle Eval/Apply pour son implémentation plus simple et ses bonnes performances dans les cas les plus courants (Marlow et Peyton-Jones, 2006). L'idée, appelée *décurryfication*, est de dédoubler la fonction app du modèle sémantique des sections précédentes en une famille de fonctions ap_n, et de représenter les fonctions n-aires à l'aide de fonctions n-aires du métalangage, comme le montre la figure 3.5. La figure 3.6 montre une interprétation adaptée aux fonctions d'arité supérieures, mais peu optimisantes :

1. $\mathsf{ap}_n\ (\mathsf{Lam}_m\ f)\ N_1 \ldots N_n = \mathsf{Lam}_{m-n}\ (f\ N_1 \ldots N_n)$
2. $\mathsf{ap}_n\ (\mathsf{Lam}_m\ f)\ N_1 \ldots N_n = f\ N_1 \ldots N_n$
3. $\mathsf{ap}_n\ (\mathsf{Lam}_m\ f)\ N_1 \ldots N_n = \mathsf{ap}_{n-m}\ (f\ N_1 \ldots N_m)\ N_{m+1} \ldots N_n$

où les conditions sont : sur (1) que $n < m$, sur (2) que $n = m$, et sur (3) que $n > m$.

Figure 3.5 Une famille d'opérateurs d'application

$$[\![x]\!] = x$$
$$[\![\underline{\lambda} x_1 \cdots x_n.\ M]\!] = \mathsf{Lam}_n\ (\lambda x_1 \cdots x_n.\ [\![M]\!])$$
$$[\![M\ \underline{\cdot}\ N_1 \ldots N_n]\!] = \mathsf{ap}_n\ [\![M]\!]\ [\![N_1]\!] \ldots [\![N_n]\!]$$

Figure 3.6 Interprétation de fonctions d'arité supérieure.

elles n'exploite que les opportunités de décurryfication manifestes dans le terme source. Il est toujours possible de faire une analyse préalable des termes telle que celle de Dargaye (2007) ou sa généralisation à l'ordre supérieur (Dargaye et Leroy, 2009), pour mieux décurryfier les termes tels que $\mathsf{ap}_2(\mathsf{ap}_2\ f\ v\ w)\ x\ y$.

Remarquons qu'il n'est nullement besoin d'un nombre infini de fonctions ap_n ni d'un nombre infini de constructeurs Lam_n. Dans la pratique, la plupart des fonctions sont de petite arité. Les fonctions d'arité supérieure à 5 ou 6 sont rares. Les fonctions d'arité supérieure peuvent toujours être curryfiées, mais à l'aide de fonctions n-aires plutôt que des fonctions unaires. Cet encodage entraîne certes un surcoût, mais un surcoût bien plus faible qu'une curryfication avec des fonctions unaires et seulement pour les fonctions de grande arité, qui représentent typiquement un faible pourcentage de toutes les fonctions dans un terme de preuve.

L'interprétation modifiée des trois fonctions ci-dessus est maintenant :

$$[\![\mathsf{nil}]\!] = \mathsf{Lam}_2\ (\lambda f\, g.\ f)$$
$$[\![\mathsf{cons}]\!] = \mathsf{Lam}_4\ (\lambda h t f\, g.\ \mathsf{ap}_2\ g\ h\ (\mathsf{ap}_2\ t\ f\ g))$$
$$[\![\mathsf{map}]\!] = \mathsf{Lam}_2\ (\lambda f\, l.\ \mathsf{ap}_2\ l\ \mathsf{nil}\ (\mathsf{Lam}_2\ (\lambda h t.\ \mathsf{ap}_2\ \mathsf{cons}\ (\mathsf{ap}_1\ f\ h)\ t)))$$

Au delà de l'économie d'allocations de structures de données en mémoire, la décurryfication permet de récolter le bénéfice de stratégies d'implémentations des fonctions du métalangage, notamment les représentations avancées des environnements dans les clôtures, telle que des structures hybrides entre tableaux et listes liées. Par exemple, la ZAM choisit de distinguer une

partie volatile et une partie persistante dans l'environnement des clôtures (Leroy, 1990). La partie volatile est une pile où sont ajoutés les arguments, à la manière des machines dont la convention d'appel est Push/Enter. La partie persistante est un tableau alloué sur le tas, permettant un accès rapide. La création d'une nouvelle clôture se fait en copiant la pile d'arguments et la partie persistante pour former un nouvel environnement. Dans le cas courant d'une fonction n-aire appliquée à n arguments d'un coup, aucune clôture intermédiaire n'est ainsi allouée. Permettre à l'environnement d'exécution du métalangage de reconnaître les fonctions n-aires comme tel, et non une suite de fonctions entrecoupée d'étiquettes, permet de récupérer le fruit de ces optimisations.

3.6.2 Compilation du filtrage

La simplicité de la généralisation du schéma d'interprétation de la section 3.3 aux règles de réécriture lui vaut un sérieux manque d'efficacité. Les termes générés par une algèbre libre de constructeurs sont si courants et si fondamentaux à la programmation que de nombreux langages offrent des mécanismes de représentation économes dédiés et un filtrage rapide (Augustsson, 1985, Le Fessant et Maranget, 2001 et Maranget, 2008). L'idée est de réutiliser ces mécanismes pour la réécriture.

La représentation des constructeurs avec l'étiquette Con est inefficace parce que cet étiquetage augmente sensiblement le taille en mémoire des structures de données du langage objet et augmente le nombre d'indirections, comme le montre la figure 3.7. L'allocation prend plus de temps et le filtrage doit déréférencer plus de pointeurs. Un constructeur du métalangage lui est représenté comme un bloc contigu de mots en mémoire commençant avec un numéro. Ce numéro sert à identifier le constructeur de manière beaucoup plus rapide qu'une chaîne de caractères qui représenterait le nom. Durant le filtrage, les champs du constructeur peuvent être accédés directement car le compilateur connaît statiquement le décalage de chacun des champs par rapport à l'adresse en mémoire du constructeur. Ainsi l'accès se fait beaucoup plus rapidement qu'en parcourant une chaîne de noeuds d'applications.

L'idée est donc de représenter les constantes impliquées dans le filtrage de règles de réécriture de manière similaire. De la même manière que nous avons généralisé app à $\mathsf{ap}_1, \ldots, \mathsf{ap}_n$ dans la section précédente, nous pouvons pour cela utiliser une famille de représentations d'applications de constantes $\mathsf{Ap}_1, \ldots, \mathsf{Ap}_n$. Si nous imposons de plus une contrainte de partage maximal des chaînes de caractères représentant les constantes, l'égalité de deux noms de constantes peut être testée en comparant simplement les pointeurs vers les chaînes de caractères — c'est-à-dire des numéros. L'application d'une constante n-aire à n arguments est représentée par

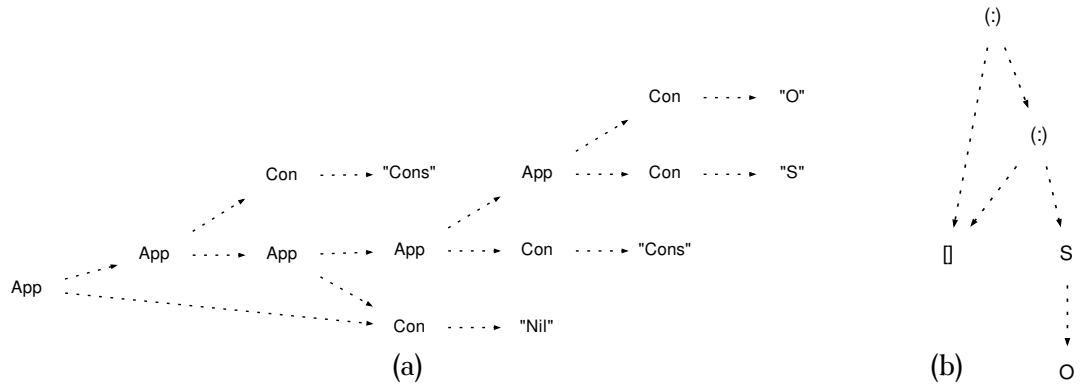


Figure 3.7 La représentation de la valeur `[[S 0]]` (la liste singleton contenant la liste singleton contenant l'entier de Peano 1) dans le tas par GHC 6.12, avec (a) un constructeur Con et (b) constructeurs natifs.

$$\mathsf{Ap_n}\ (\mathsf{Con}\ c)\ M_1 \ldots M_n.$$

Si des règles de réécriture sont associées à une constante, l'arité de celle-ci est fixée par la forme du membre gauche des règles de réécriture. Si aucune règle de réécriture n'est associée à une constante, alors l'arité de celle-ci peut-être déterminée par le côté gauche d'une règle de réécriture où la constante apparaît dans un motif. Ainsi le système de règles ci-dessous nous apprend que les constantes O,S sont d'arité 0,1, respectivement :

$$[y : \mathsf{nat}] \qquad \mathsf{plus}\ \mathsf{O}\ y \longrightarrow y.$$

$$[x : \mathsf{nat}, y : \mathsf{nat}]\ \mathsf{plus}\ (\mathsf{S}\ x)\ y \longrightarrow \mathsf{S}\ (\mathsf{plus}\ x\ y).$$

x,y ne sont pas des constantes mais des variables de motif puisque liées localement dans le contexte des règles.

Cette optimisation ne limite pas la forme des règles de réécriture. Mais si deux règles de réécriture utilisent la même constante à deux arités différentes dans leurs motifs, alors nous sommes obligés de prendre l'arité la plus petite comme arité de la constante. Ceci parce que la règle (η) n'est pas admissible dans le $\lambda\Pi$-calcul modulo. Dans un système où (η) fait partie de la conversion[21], $\mathsf{S} : \mathsf{nat} \to \mathsf{nat}$ est équivalent à $\lambda x.\ \mathsf{S}\ x : \mathsf{nat} \to \mathsf{nat}$. On peut donc substituer l'interprétation de la deuxième forme à celle de la première partout où il manque l'argument à S, si l'on choisit 1 comme l'arité de cette constante.

Remarque 3.3 L'arité d'une constante peut aussi parfois être extraite du seul type de celle-ci, sans regarder les règles de réécriture. En $\lambda\Pi$-modulo, l'arité d'une constante n'est pas toujours évidente car les calculs sont autorisés au niveau des types :

[21] LF (Harper et Licata, 2007) en est un exemple.

$$\mathsf{NatToNat} : \mathbf{Type}.$$

$$[]\ \mathsf{NatToNat} \longrightarrow \mathsf{nat} \to \mathsf{nat}.$$

$$\mathsf{plus} : \mathsf{nat} \to \mathsf{NatToNat}.$$

Ici l'arité de plus n'est pas évidente statiquement. Mais on peut apporter une borne inférieure à l'arité de plus : cette constante est au moins d'arité 1. Si la constante plus est toujours utilisé à une arité 2 dans les motifs des règles de réécriture, alors l'interpréter comme une constante d'arité 1 demandera plus de noeuds en mémoire pour représenter les applications à deux arguments et le filtrage sera moins performant.

Le $\lambda\Pi$-modulo est un cadre très général et peu structuré. Les constructeurs et les récurseurs de types inductifs y sont représentés à l'aide de constantes et le comportement calculatoire des récurseurs est donné sous forme de règles de réécriture (voir la section 5.2). Ces règles et ces constantes sont perdues parmi toutes les autres constantes et règles de réécriture à portée de main, mais les règles de réécriture et le type des constantes issus d'encodages de types inductifs se conforment à des invariants plus forts que les règles et constantes quelconques. En particulier les constantes qui correspondent à des constructeurs apparaissent toujours saturées (avec même arité) dans tous les termes et les motifs. De plus les ensembles de constructeurs de deux types inductifs sont toujours disjoints. On peut donc directement utiliser les constructeurs du métalangage comme interprétation des constructeurs du langage encodé dans le $\lambda\Pi$-modulo. À un constructeur c d'arité n du langage objet correspond un constructeur c de même arité du métalangage. Nous arrivons ainsi à des représentations en mémoire comme celle de la partie (b) de la figure 3.7.

Dans un métalangage statiquement typé où les types de données sont fermés, au sens où il n'est pas possible de rajouter des constructeurs à un type *a posteriori*, inventer de nouveaux constructeurs du métalangage pour chaque nouvelle constante dans le langage source est problématique pour la compilation modulaire des développements en $\lambda\Pi$-modulo. Il faut en effet connaître *a priori* tous les constructeurs dont on aura besoin avant de déclarer le type du métalangage qui sera l'univers de toutes les interprétation de termes du $\lambda\Pi$-modulo comme le type suivant en Haskell :

```
data Term = F Int
          | Lam (Term -> Term)
          | App Term Term
          | O | S Term | Nil | Cons Term Term | ...
```

Une solution est d'employer un des divers mécanismes de types extensibles tels que le type des exceptions exn de Standard ML (le seul type admettant l'ajout de nouveaux constructeurs à la

volée) ou les variantes polymorphes (Garrigue, 1998) de OCaml. À défaut, si la performance est réellement primordiale, on peut ajouter $n \times m$ constructeurs de la forme

$$\texttt{Con}_n \underbrace{\texttt{Term} \ldots \texttt{Term}}_{m \text{ fois}}$$

qui serviront pour représenter les n premiers constructeurs d'arité inférieure ou égale à m de chaque type inductif.

En résumé, la bonne solution est fonction de l'environnement d'exécution sélectionné pour normaliser les termes du $\lambda\Pi$-modulo. Et encore une fois, l'objectif ici est de se tirer d'affaire en réutilisant des plate-formes existantes sans modifications tout en observant que la pénalité éventuelle imposée par cette contrainte est minime — voir négligeable dans certains cas comme nous le verrons dans la section 3.8.

3.7 Extension aux termes du calcul des constructions inductives

Afin de valider l'approche pour le test de conversion de théorie des types dépendantes présenté dans les sections précédentes par rapport à la machine virtuelle de Grégoire et Leroy (2002) implémentée dans Coq, nous avons été amené à porter la technologie de normalisation de Dedukti à Coq. Nous avons développé une branche expérimentale de Coq où le test de conversion est implémenté par compilation des termes à comparer vers des programmes OCaml pour les termes clos dont le type est un type de base. Dénès (2010) a ensuite généralisé cette implémentation à tous les termes Coq ouverts et de n'importe quel type, tout en améliorant les temps de compilation des termes, avant de passer à un schéma de représentations optimisé pour OCaml mais moins portable. Ce travail requit d'adapter et de généraliser l'approche aux formes syntaxiques et règles de réductions spécifiques au calcul des constructions inductives (CIC). Cette section montre comment étendre la normalisation par évaluation non typée au CIC avant de passer aux comparatifs de performance dans la section 3.8. Nous ne traiterons ici que l'analyse par cas du CIC et les points fixes, les autres formes syntaxiques étant immédiates.

3.7.1 Interprétation du filtrage

Les formes `match ... with ... end` de Coq[22] ne peuvent être représentées directement comme des constructions de filtrage du métalangage. Dans le cas où l'analyse par cas se fait sur un terme dont la tête n'est pas un constructeur, la règle de réduction (ι) du CIC ne s'applique

[22] Qui sont du sucre syntaxique pour l'analyse de cas sans motifs de la forme $\mathbf{case}_I(M_s, M_p, M_1|\ldots|M_n)$ du CIC, où $M_1, \ldots, M_n$ sont le corps de chacune des branches correspondant à chacun des constructeurs du type inductif I, et M_p est le prédicat donnant le type de retour de l'analyse de cas.

$$[\![\mathbf{case}_I(M_s,M_p,M_1|\ldots|M_n)]\!] = \mathsf{app}\left(\begin{pmatrix}\mathbf{fix}\ (\lambda f.\ \lambda s.\ \mathbf{case}\ s\ \mathbf{of} & & \\ C_1 & \to & [\![M_1]\!] \\ \vdots & & \vdots \\ C_n & \to & [\![M_n]\!] \\ \mathbf{default} & \to & \mathsf{Case}\ s\ [\![M_p]\!]\ f\ )\end{pmatrix}\right)[\![M_s]\!]$$

Figure 3.8 Interprétation des analyses de cas d'un type inductif I du CIC, où $C_1, \ldots, C_n$ sont les n constructeurs de I.

pas plus que d'autres règles de réduction, de sorte qu'il est nécessaire de générer une forme neutre représentant une analyse par cas bloquée. Pour éviter de générer deux représentations pour chaque analyse de cas, ce qui provoquerait une explosion exponentielle de la taille de l'interprétation lorsque les analyses par cas sont imbriquées, l'interprétation est donnée (figure 3.8) comme une fonction récursive que l'on pourra réutiliser plus tard pour la réification de chaque branche de l'analyse de cas.

Nous avons besoin d'une étiquette supplémentaire : le constructeur Case pour représenter ces analyses de cas bloquées. Durant la réification d'une analyse de cas d'une valeur de type I, on obtient une réification de chacune des branches en donnant chacun des constructeurs de I à la fonction de filtrage, comme suit :

$$\downarrow_n \mathsf{Case}\ M_s\ M_p\ f\ = \underline{\mathbf{case}}_I(\downarrow_n M_s, \downarrow_n M_p, \downarrow_n f\ C_1|\ldots|\downarrow_n f\ C_n)$$

3.7.2 Points fixes

Un point fixe n-aire du CIC désigne le n-ième argument comme décroissant structurellement à chaque appel récursif. Par simplicité, nous ne traitons dans cette section que les points fixes unaires, considérant les points fixes n-aires comme du sucre syntaxique[23] pour

$$\underline{\lambda} x_1.\ \cdots \underline{\lambda} x_{n-1}.\ \underline{\mathbf{fix}}_1\ (f\ :A := M\,).$$

L'interprétation des points fixes est donnée de façon très naturelle en fonction d'un opérateur de points fixes termfix supprimant le dépliage du point fixe dans le cas où la tête de l'argument donné au point fixe n'est pas un constructeur :

$$[\![\underline{\mathbf{fix}}_1\ (f\ :A := M\,)]\!] = \mathsf{termfix}\ [\![A]\!]\ [\![M\,]\!]$$

[23] Par remontée ou descente de abstractions et des définitions locales (« lambda lifting » (Johnsson, 1985) et « let floating » (Peyton Jones et al., 1996)) si besoin. En notant que les termes acceptés par Coq sont déjà dans une forme ne demandant pas de telles transformations.

L'opérateur termfix est donné par les équations récursives suivantes :

$$\begin{aligned}
&\mathsf{termfix}\ A\ M\ x = \mathsf{Fix}\ A\ M && \text{si } x = \mathsf{F}\ n \\
&\mathsf{termfix}\ A\ M\ x = \mathsf{ap}_2\ M\ (\mathsf{Lam}\ (\lambda y.\ \mathsf{termfix}\ A\ M\ y))\ x && \text{sinon}
\end{aligned}$$

termfix est donc une fonction d'emballage autour de l'opérateur de points fixes natif du métalangage. La réification consiste à déplier la fonction dont on a pris le point fixe :

$$\downarrow_n\ \mathsf{Fix}\ A\ M = \underline{\mathbf{fix}}_1\ (n : (\downarrow_n A) := \downarrow_{n+1} \mathsf{app}\ M\ (\mathsf{F}\ n))$$

La généralisation aux points fixes mutuels ne pose pas de difficultés particulières.

3.8 Comparatif de performances

3.8.1 Micro-tests

L'usage que l'on fait de la NbE non typée dans Dedukti est celui d'une machinerie bon marché pour faire le test de conversion de théories des types dépendantes. Nous examinons dans cette section les performances de cette machinerie ; nous saurons ainsi si cette machinerie au rabais en terme d'effort d'implémentation est à forte valeur ajoutée. Nous commencerons par mesurer l'effet des diverses optimisations présentées sur de petits exemples, comparant d'une part ces résultats avec un évaluateur, mais aussi avec la variante de normalisation par évaluation non typée au coeur du simplificateur de Isabelle (Aehlig et al., 2008). Les chiffres de l'évaluateur nous donnent un plancher que l'on ne peut espérer dépasser sans baser notre algorithme de normalisation sur une autre technologie qu'un évaluateur trouvé tel quel sur l'étagère. Nous utilisons Template Haskell pour l'implémentation et GHC 6.10 comme évaluateur. Nous comparerons dans un deuxième temps la NbE non typée étendue au CIC avec la machine virtuelle de Grégoire et Leroy (2002) sur un gros développement faisant un usage intensif de la réflexion à grande échelle.

Voici les 6 variantes de NbE comparées :

ahn Il s'agit de la NbE non typée de Aehlig et al. (2008). Toutes les fonctions sont appliquées à des arguments un à un. Tous les arguments à toutes les fonctions sont emmagasinés dans des listes en ordre inversé, ce qui permet de coder de manière uniforme les fonctions d'arité supérieure.

singlearity L'interprétation de la section 3.3.

evalapply L'interprétation optimisée de la section 3.6.1.

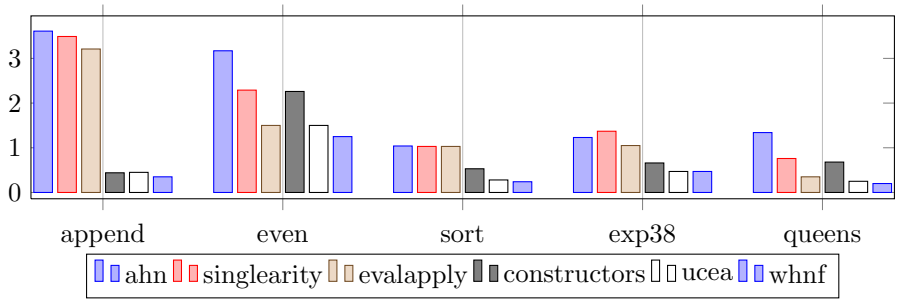


Figure 3.9 Représentation visuelle des résultats de la table 3.1.

flavor	append	%	even	%	sort	%	exp3-8	%	queens	%
ahn	3.61	1031	3.17	253	1.04	433	1.23	261	1.34	670
singlearity	3.49	997	2.29	183	1.03	429	1.37	191	0.76	380
evalapply	3.21	917	1.50	120	1.03	429	1.05	223	0.35	175
constructors	0.44	125	2.26	180	0.53	220	0.66	140	0.68	340
ucea	0.45	128	1.50	120	0.28	116	0.47	100	0.25	120
whnf	0.35	100	1.25	100	0.24	100	0.47	100	0.20	100

Tableau 3.1 Temps absolus d'exécution et
temps normalisés par rapport au temps de whnf.

constructors Les constructeurs du langage objet deviennent des constructeurs du métalangage, comme dans la section 3.6.2.

ucea Combinaison de l'interprétation « evalapply » et « constructors ».

whnf L'interprétation identité, où les termes du langage objet sont interprétés par eux-mêmes. Avec cette interprétation, on ne peut pas normaliser de termes ouverts et la réification peut être définie que pour les termes typables par un type de base.

Nous comparons la performance de ces variantes sur les programmes suivants :

append Concaténation de deux grosses listes d'entiers de taille 50,000.

even Décide la parité de la taille de la liste donnée en argument. Les listes sont représentées ici comme des encodages de Church. Il n'y a donc aucun filtrage, de sorte que ce test mesure purement la performance des β-réductions.

sort Tri d'une grosse liste d'entiers encodée avec des constructeurs. Le filtrage joue ici un rôle plus important. L'implémentation du tri est celle de la bibliothèque standard de Haskell.

exp3-8 Programme test de la suite de tests nofib (Partain, 1992) calculant l'entier unaire de Peano $3^8 = 6561$.

queens Énumère toutes les solutions du classique problème de satisfaction de contraintes suivant : placer 10 dames sur un échiquier de taille 10x10 en évitant qu'une des dames puisse en attaquer une autre.

La figure 3.9 donne les résultats. Notons tout de suite que la grande majorité de l'amélioration du temps d'exécution est dû à l'interprétation des constructeurs comme des constructeurs, confirmant la place centrale qu'occupent les structures de données — ici en programmation mais plus tard dans les termes de preuves également. Une analyse des traces du tas durant l'exécution des programmes ci-dessus montre que l'usage de constructeurs permet moitié moins d'allocations de d'objet en mémoire que sans cette optimisation, ce qui permet d'alléger la pression sur le ramasse-miettes, dont le travail compte pour une part significative des temps d'exécution.

Curryfier plutôt que grouper les arguments dans des listes fréquemment construites puis déconstruites au fil des créations de clôtures et des applications, permet un gain dans la plupart des tests. La décurrification des fonctions n-aires permet d'aller plus loin encore, offrant des temps d'exécution moitié moindres sur les tests où les applications prédominent. Les gains en performance augmentent aussi sur les tests avec des fonctions de plus grande arité, comme « queens » et son usage intensif de catamorphismes comme foldr.

Moyennant quelques optimisations très simples à mettre en oeuvre, nous obtenons un normaliseur dont la performance est proche de celle de l'évaluateur sous-jacent. Dans des tests pathologiques n'impliquant aucun filtrage mais seulement des β-réductions, comme « even », nous observons une pénalité dû à l'étiquetage des termes de l'ordre de 20%. Ces résultats sont obtenus avec une stratégie d'évaluation paresseuse. L'impact de l'étiquetage après compilation par d'autres compilateurs moins optimisants ou choisissant d'autres stratégies d'évaluation pourrait s'avérer plus ou moins lourd.

3.8.2 Vérification d'une preuve réflexive dans Coq

Les preuves réflexives cherchent à minimiser la taille de la preuve et le travail de l'utilisateur au prix d'un calcul plus intensif pendant la vérification de la preuve. Pour certaines preuves, le temps de calcul domine très largement le temps total de vérification. Ces preuves tirent un bénéfice particulier d'un test de conversion implémenté comme une normalisation par évaluation car le temps de calcul est si grand que le temps de compilation nécessaire est largement amorti par une décision du test de conversion plus rapide. Nous avons testé la normalisation par évaluation sur un procédure d'élimination des quantificateurs pour l'arithmétique de Presburger de Joshi et Mahboubi (2009).

La table 3.2 montre l'explosion du temps nécessaire en fonction de n pour vérifier la formule suivante en arithmétique de Presburger :

$$\exists x_1, \ldots, x_n.\ \forall y,z \in \{x_1, \ldots, x_n\}.\ y \neq z \wedge y \leq n,$$

variables	1	%	2	%	3	%	4	%	5	%
no conv	0.63	94	0.68	94	1.40	93	2.25	77	3.92	3.11
NbE	0.64	95	0.70	97	1.42	94	2.30	79	45.48	33
Coq VM	0.67	100	0.72	100	1.50	100	2.92	100	136.2	100

Tableau 3.2 Résolution du formules de n variables avec la procédure d'élimination des quantificateurs de Cooper.

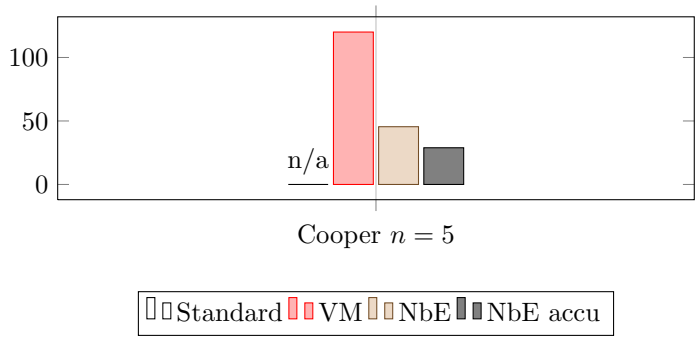


Figure 3.10 Élimination des quantificateurs de Cooper avec la réduction standard, la VM, la NbE et la réduction de Dénès (2010).

où $x_1, \ldots, x_n$ sont des entiers strictement positifs. À partir de $n = 6$, le problème devient si difficile que l'exécution de l'algorithme épuise toute la mémoire vive disponible après environ 30 minutes. Le temps de référence est ici le temps nécessaire à la vérification en utilisant la machine virtuelle existante de Coq. Le temps plancher que nous ne pouvons espérer battre est le temps de vérification de la preuve où le test de conversion est la fonction constante qui répond toujours « vrai ». Ce chiffre nous donne le temps de vérification hors test de conversion.

La dernière colonne nous montre que le test de conversion occupe presque tout le temps de vérification et que la normalisation par évaluation présentée ici offre un gain en performance par rapport à la machine virtuelle de l'ordre de 3. Mais d'après des mesures faites par Dénès que l'on retrouve dans la figure 3.10, l'interprétation des termes présentée dans ce chapitre est peut être améliorée de manière significative lorsque le langage de l'interprétation des termes est OCaml. L'algorithme de normalisation est similaire, mais Dénès (2010) montre qu'il est possible d'utiliser des fonctions primitives dérobées[24] permettant de diriger la normalisation par la forme des clôtures dans le tas, sans recours à des étiquettes explicites. On passe ainsi à un gain d'un facteur 4. Le test de conversion par défaut de Coq ne parvient pas quant à lui à calculer le résultat final, faute de ressources mémoire.

[24] En particulier `Obj.magic` et `Obj.repr`.

3.9 Conclusion

Tout comme passer d'un interpréteur naïf à un compilateur optimisant signifie passer de l'intraitable au faisable pour l'évaluation de nombreux programmes, la compilation de parties coûteuses de vérifications de preuves peut rapporter d'énormes bénéfices. (Grégoire et Leroy, 2002) ont montré que recycler des technologies de compilation existantes pour vérifier les preuves était viable. Les divers auteurs de normalisation ont montrée qu'il était possible de recycler sans même ouvrir le capot du compilateur. Nous avons montré dans ce chapitre que se refuser à ouvrir le capot ne signifiait pas pour autant une grosse pénalité de performances. En remarquant que la normalisation par évaluation est une auto-réduction de Mogensen avec un schéma de représentation légèrement adapté, nous sommes arrivés à un auto-réduiseur très naturel et plus efficace que l'auto-réduiseur originel de Mogensen (1992). Nous avons ensuite montré que cet auto-réduiseur, un algorithme de normalisation par évaluation non typée, se généralisait très naturellement à une relation de réduction donnée par ajout de règles de réécriture à $(\longrightarrow_\beta)$, et encaissait tout aussi naturellement l'ajout dans le langage objet de constructions d'analyses de cas et d'opérateurs de points fixes avec conditions de garde.

Chapitre 4
Systèmes de types purs hors contexte

Le chapitre 2 a présenté divers systèmes de types pour les termes du λ-calcul. Par la correspondance de Curry-Howard, un terme $M : A$ est une preuve de A que l'on peut considérer comme une formule. Vérifier que M est bien une preuve de A revient à vérifier que M est typable et que le type de M est convertible à A. Nous présentons dans ce chapitre une représentation des termes de preuve à vérifier admettant un algorithme de vérification très simple pour une large classe de systèmes de types (les Pure Type Systems de Barendregt (1991)). Nous discutons également de l'intérêt d'une telle représentation des termes pour la conception d'une interface entre l'algorithme de vérification et le monde extérieur.

4.1 Les jugements de typage comme des clôtures

La difficulté majeure de la vérification de termes de preuves typés par un PTS réside en la nécessité de décider la convertibilité de deux termes durant le typage. Nous avons vu dans le chapitre 3 comment décider cette convertibilité de manière sûre et efficace en réutilisant les mécanismes d'évaluation du langage d'implémentation de l'algorithme de décision. Pour rappel, nous avions défini le type des termes comme suit :

```
data Term = Var Name
          | Lam (Term -> Term)
          | App Term Term
```

Les lieurs du langage objet sont représentés par des fonctions dans le métalangage. Les applications étaient elles aussi représentées par une application du métalangage, modulo le plongement profond du langage objet. Puisque, pour les besoins de la procédure décidant la convertibilité, les fonctions du langage objet sont représentées à l'aide de fonctions du métalangage, il est naturel de se demander si les termes du langage objet ne pourraient pas réutiliser cette idée à bon escient pour la vérification des types. La représentation des termes à typer pourrait ainsi être la représentation des termes à convertir — un seul type de données serait alors nécessaire.

Les algorithmes d'inférence de type du chapitre 2 utilisent une structure d'environnement pour maintenir en mémoire les hypothèses de typage pour chacune des variables libres d'un terme. Cette structure est reflétée dans le jugement de typage du système de type dans lequel

se place l'algorithme d'inférence. Un terme M est de type A sous les hypothèses de typage contenues dans Γ :

$$\Gamma \vdash M : A$$

Après lecture du chapitre 1, cette forme devrait paraître familière : puisque $\mathcal{FV}(M) \subseteq \mathsf{dom}(\Gamma)$, un jugement est en fait une clôture, c'est à dire la combinaison d'un terme et d'un environnement associant une sémantique à chacune des variables libres du terme.

Les langages de programmation fonctionnelle sont des choix particulièrement appropriés de langages d'implémentation pour un système de manipulation symbolique tels que le sont les assistants interactifs de preuve, et *a fortiori* les systèmes de vérification de preuves. La très grande majorité des assistants de preuve d'aujourd'hui sont implémentés dans des langages fonctionnels, tels que HASKELL (AGDA, EPIGRAM), divers dialectes de ML (COQ, HOL LIGHT, ISABELLE) ou encore dans des variantes de LISP (ACL2, MINLOG, PVS). Seul MIZAR fait figure d'exception. Tous ces langages d'implémentation sont basés sur le λ-calcul. Les fonctions y sont de ce fait des citoyennes de première classe — comme tous les autres types de valeurs, les fonctions peuvent être passées comme argument à d'autres fonctions et il est possible d'écrire des fonctions qui rendent une fonction comme valeur de retour. Opérationnellement, dans tous ces langages les fonctions correspondent à des clôtures ; une construction associant au code de la fonction un environnement dans lequel évaluer ce code. C'est cette construction de clôture qui permet à une fonction de conserver son environnement d'origine, même lorsque celle-ci est appelée dans le corps d'une fonction étrangère, comme dans l'exemple suivant :

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map xs

increment :: Int -> [Int] -> [Int]
increment n xs = let g x = x + n in map g xs
```

Ici la fonction `g` a une variable libre `n` liée dans `increment`. Durant l'évaluation de l'expression `increment 1 []`, l'argument passé à `map` s'évaluera en une clôture contenant le code de `g` et aussi un environnement associant 1 à la variable libre `n`. Ce modèle d'implémentation — toutes les valeurs[25] sont des clôtures — est d'ailleurs le *modus operandi* du schéma de compilation décrit dans la section 1.5 du chapitre 1.

Si le système de type admet la propriété de l'unicité du typage, un algorithme de typage est une fonction qui accepte en entrée le contexte de typage Γ et le terme M de $\Gamma \vdash M : A$, et

[25] Sauf quelques types primitifs tels que les entiers.

produit A en valeur de retour. L'idée est donc d'écrire un algorithme de typage qui représente la clôture (Γ, M) par une clôture du métalangage. Ce faisant, l'algorithme de typage ne nécessiterait plus la déclaration d'un type abstrait des contextes de typage et déléguerait les opérations usuelles sur ce type abstrait, tels que `lookup` et `push`, à des constructions du métalangage. La correction de l'algorithme de typage ne dépendrait plus de la correction des opérations sur le type abstrait des contextes. En contrepartie, la propriété d'adéquation entre les termes typés du λ-calcul et leur encodage dans le métalangage serait potentiellement moins évidente. L'algorithme serait plus court, plus simple et hériterait directement de la bonne performance de l'implémentation des environnements dans le métalangage. Que l'environnement d'une clôture (ou bien le contexte de typage) soit représenté comme une liste chaînée ou un tableau, ou encore par une structure hybride (Leroy, 2005) est un détail d'implémentation du métalangage qui pourra être changé à volonté, pour des raisons de performance notamment.

Avant de présenter cet algorithme, nous introduisons une série de systèmes de types de plus en plus puissants, qui auront tous pour point commun l'absence de contexte de typage dans les jugements ; ceux-ci seront tous de la forme

$$\vdash M : A$$

Nous dirons que nous avons là des systèmes de type « hors contexte », en référence aux grammaires hors contexte en théorie des langages. Chacun de ces systèmes de types admettra un algorithme de typage lui aussi hors contexte, dont la correction et la complétude seront établis.

4.2 Typage hors contexte pour le λ-calcul simplement typé

Reprenons l'algorithme d'inférence du type d'un terme de la section 2.6 du chapitre 2. Cet algorithme est correct et complet par rapport aux dérivations de typage des termes du λ-calcul simplement typé. Il peut donc être utilisé comme procédure de décision pour la typabilité d'un terme. Nous étudions dans cette section un autre algorithme pour une autre représentation des termes.

Définition 4.1 (λ_b^{st}-calcul) Les termes du λ_b^{st}-calcul sont caractérisés inductivement par la grammaire en forme Backus-Naur suivante :

$$\begin{array}{rcll} \mathsf{Var} & \ni & x, y, z & \\ \mathsf{Type} & \ni & \sigma, \tau & ::= \quad \iota \mid \sigma \to \tau \\ \mathsf{Term_b} & \ni & M, N & ::= \quad x \mid \lambda x : \tau \,.\, M \mid M\ N \mid [x : \tau] \end{array}$$

$$(cast_b)\ \frac{}{\vdash [x:\tau]:\tau} \qquad (abs_b)\ \frac{\vdash \{x/[x:\sigma]\}M:\tau}{\vdash \lambda x:\sigma.\ M:\sigma\to\tau}$$

$$(app_b)\ \frac{\vdash M:\sigma\to\tau \qquad \vdash N:\sigma}{\vdash M\ N:\tau}$$

Figure 4.1 Système de type hors contexte pour le λ-calcul simplement typé.

Ce langage est une extension du λ-calcul avec une construction supplémentaire $[x:\tau]$ de « boîte » permettant de mémoïser le type d'une variable. Nous qualifierons également ces boîtes d'*annotations de type*.
L'opération de substitution d'un terme pour une variable est définie comme pour le λ-calcul (où l'on suppose $x \neq y$) :

$$\begin{aligned} \{N/x\}x &= N \\ \{N/x\}y &= y \\ \{N/x\}(\lambda x:\tau.\ M) &= \lambda x:\tau.\ M \\ \{N/x\}(\lambda y:\tau.\ M) &= \lambda y:\tau.\ \{N/x\}M \qquad \text{if } y\notin\mathcal{FV}(N) \\ \{N/x\}(M\ M') &= \{N/x\}M\ \{N/x\}M' \\ \{N/x\}[x:\tau] &= N \\ \{N/x\}[y:\tau] &= [y:\tau] \end{aligned}$$

Les notions de clôture par contexte d'une relation de réduction et de α-conversion sont étendues de la manière usuelle aux termes du λ_b^{st}-calcul. La β-réduction est la même que pour le λ-calcul.

Ainsi équipés de la possibilité d'annoter certaines variables d'un terme, nous pouvons énoncer un système de type pour le λ_b^{st}-calcul où il n'est nullement question de contexte de typage. Les règles de déduction de ce système de type sont données dans la figure 4.1. Dans ce système, les variables ne sont pas typables, puisqu'à tout instant l'ensemble des hypothèses de typage est vide. Cependant, la règle pour les abstractions postule que si une variante du corps de l'abstraction où toutes les occurrences libres de la variable abstraite apparaissent décorées d'une annotation de type, est de type τ, alors l'abstraction entière est de type $\sigma\to\tau$.

4.2.1 Complétude du système de type hors contexte

Nous prouvons dans cette section que tout terme du λ-calcul typable dans le système de type de la section 2.1 du chapitre 2 l'est aussi dans le système de type présenté dans la figure 4.1. Dans ce qui suit, un jugement de la forme $\vdash M:\tau$ est un jugement du système de type sans environnement de la figure 4.1. Nous écrirons $[]\vdash M:\tau$ si, dans le contexte vide, M est typable par τ dans le système de la section 2.1 du chapitre 2, pour éviter toute ambiguïté.

Nous commençons par observer que les captures de variables libres dans les dérivations de typage sont impossibles.

Lemme 4.2 *Pour toute variable x et type τ, $\mathcal{FV}([x:\tau]) \subseteq \mathcal{FV}(x)$.*

Preuve. Immédiat. □

Nous n'avons donc pas besoin ici de considérer les termes modulo α-conversion, bien que les règles de typage fassent usage de l'opération de substitution, ni de principes de raisonnement puissants tels que la convention de Barendregt, puisqu'une substitution de la forme $\{x/[x:\tau]\}$ n'entraînera jamais de capture de variables libres. La formalisation de la preuve en est grandement simplifiée[26].

Définition 4.3 (Déchargement d'hypothèses de typage) Soit la fonction suivante :

$$[] \searrow M = M$$
$$\Gamma, x:\tau \searrow M = \{[x:\tau]/x\}(\Gamma \searrow M)$$

Définition 4.4 (Désannotation) Nous définissons également la fonction $(\cdot)_{\cdot}^{-1} : \mathsf{Var} \to \mathsf{Term_b} \to \mathsf{Term}$ qui supprime toutes les annotations sur une variable y :

$$(x)_y^{-1} = x \qquad (M\ N)_y^{-1} = (M)_y^{-1}\ (N)_y^{-1}$$
$$(\lambda y:\tau.\ M)_y^{-1} = \lambda y:\tau.\ M \qquad ([y:\tau])_y^{-1} = y$$
$$(\lambda x:\tau.\ M)_y^{-1} = \lambda x:\tau.\ (M)_y^{-1} \quad \text{si } y \neq x \qquad ([x:\tau])_y^{-1} = [x:\tau] \quad \text{si } y \neq x$$

Lemme 4.5 *Pour tout terme M, variables x,x' tel que $x \neq x'$ et types τ,τ' :*

1. $\{x/[x:\tau]\}\{x/[x:\tau']\}M = \{x/[x:\tau]\}M$ *(la substitution est idempotente) ;*
2. $\{x/[x:\tau]\}\{x'/[x':\tau']\}M = \{x'/[x':\tau']\}\{x/[x:\tau]\}M$;
3. $\{x/[x:\tau]\}(M)_{x'}^{-1} = (\{x/[x:\tau]\}M)_{x'}^{-1}$;
4. $(\{x/[x:\tau]\}M)_x^{-1} = (M)_x^{-1}$;
5. $\{x/[x:\tau]\}(M)_x^{-1} = \{x/[x:\tau]\}M$.

Preuve. Par induction sur M. □

Lemme 4.6 *Pour toutes variables x,x' tel que $x \neq x'$, types τ,τ' et hypothèse Γ :*

[26] Un script de preuve Coq/SSreflect avec les preuves complètes de cette section est disponible à l'adresse suivante : `http://www.lix.polytechnique.fr/~mboes/projects/contextfree/stlc.v`

1. Γ,$x' : \tau' \searrow x = \Gamma \searrow x$;
2. Γ,$x : \tau \searrow x = [x : \tau]$.

Ces égalités tiennent aussi pour les annotations :

3. Γ,$x' : \tau' \searrow x : \tau = \Gamma \searrow x : \tau$;
4. Γ,$x : \tau' \searrow x : \tau = [x : \tau']$.

Preuve. Les deux égalités se prouvent par induction sur Γ. Dans le cas inductif où $\Gamma = \Gamma'$,$y : \sigma$, si $y = x$ alors on applique la partie 1 du lemme 4.5 et l'hypothèse d'induction, sinon la partie 2 du lemme 4.5 et l'hypothèse d'induction. □

Théorème 4.7 (Complétude) *Pour tout terme M sans annotations, contexte de typage Γ et type τ, si $\Gamma \vdash M : \tau$ alors $\vdash (\Gamma \searrow M) : \tau$.*

Preuve. Par induction sur la dérivation de $\Gamma \vdash M : \tau$.

cas (var) : conséquence de la partie 1 du lemme 4.6.

cas $(weak)$: conséquence de la partie 2 du lemme 4.6.

cas (abs) : alors par inversion du jugement de typage, M est de la forme $\lambda x : \sigma.\ M'$. Nous avons

$$\vdash \Gamma,x : \sigma \searrow M' : \tau$$

et cherchons à démontrer

$$\vdash \Gamma \searrow \lambda x : \sigma.\ M' : \sigma \to \tau\,. \tag{4.1}$$

Il nous faut faire commuter le déchargement sous l'abstraction pour pouvoir construire une dérivation pour ce jugement à l'aide de la règle (abs_b). Nous observons que

$$\Gamma \searrow \lambda x : \sigma.\ M' = \lambda x : \sigma.\ (\Gamma,x : \sigma \searrow M')^{-1}_x \tag{4.2}$$

si M' ne contient pas d'annotations, par induction sur Γ et les égalités du lemme 4.5. Nous pouvons donc faire commuter le déchargement à l'aide de cette égalité, appliquer la règle (abs_b) et enfin obtenir une dérivation de la (proposition 4.1) en utilisant les égalités 5 et 2 du lemme 4.5 et par la définition de $(\searrow)$.

cas (app) : alors par inversion du typage, M est de la forme $M'\ N'$. Nous avons des dérivations de

$$\vdash \Gamma \searrow M' : \sigma \to \tau \text{ et } \vdash \Gamma \searrow N' : \sigma.$$

à partir desquelles nous pouvons construire une dérivation de

$$\vdash \Gamma \searrow M'\ N' : \tau$$

en appliquant la règle (app_b), puis en faisant commuter l'application et le déchargement, par induction sur Γ.

□

4.2.2 Correction du système de type hors contexte

Nous cherchons à prouver dans cette section qu'à chaque fois que $\vdash M : \tau$, alors il existe une dérivation de $\Gamma \vdash M' : \tau$, où Γ est formé à partir des annotations dans M et M' est obtenu par suppression des annotations de type de M.

Cependant, cette propriété est fausse en général. Voici un contre-exemple :

$$[x : \tau \to \tau]\ [x : \tau]$$

Bien que ce terme soit typable dans le système de types hors contexte, le terme $x\ x$, obtenu par effacement des annotations, n'est pas typable dans le λ-calcul simplement typé, quel que soit le contexte de typage.

Mais nous observons que si nous considérons que deux variables avec le même nom mais deux annotations de types différentes sont en fait deux variables distinctes, alors le contre-exemple ci-dessus n'en est plus un. Ce point de vue revient à considérer que le type d'une variable fait partie intégrante de son nom. Autrement dit, étant donné une fonction $(\cdot)^\bullet : \mathsf{Term_b} \to \mathsf{Term_b}$ qui sature toutes les occurrences de variables avec une annotation de type, la propriété de correction que nous chercherons à prouver dans cette section mettra en relation des termes dont les noms de variables sont des symboles avec des termes dont les noms de variables sont pris de l'ensemble formé par le produit cartésien des deux ensembles dénombrables Var et Type :

$$\mathsf{VarType} \mathrel{\hat{=}} \{(x,\iota), (x,\iota \to \iota), \ldots, (y,\iota), (y,\iota \to \iota), \ldots, \ldots\}$$

Nous noterons (x,τ) par x^τ.

Définition 4.8 ($\lambda^{\mathsf{st}}_{x\tau}$-calcul) Les termes du $\lambda^{\mathsf{st}}_{x\tau}$-calcul sont définis inductivement par la grammaire suivante :

$$\begin{array}{rcll} \mathsf{VarType} & \ni & x^\tau, y^\tau, z^\tau & \\ \mathsf{Term_x} & \ni & M, N \qquad ::= & x^\tau \mid \lambda x^\tau : \sigma.\ M \mid M\ N \end{array}$$

De façon équivalente, étant donné une bijection entre Var et $\mathbb{N}$, nous aurions pu laisser la classe des noms de variables inchangée en établissant une numérotation de Gödel

$g : \mathsf{Var} \times \mathsf{Type} \to \mathbb{N}$. Peu importe le procédé, cette internalisation du type dans le nom des variables est non sans rappeler la logique des systèmes formels de la famille HOL (HOL4, HOL Light, ProofPower, Isabelle), où toutes les occurrences de variables sont systématiquement accompagnées de leur type et où deux variables aux types distincts sont distinctes. Nous en rediscuterons dans la section 4.3.

Définition 4.9 (Saturation des annotations) Dans le λ_b^{st}-calcul, une annotation sur une occurrence de variable est optionnelle. Pour tout terme M, le terme $M^\bullet$ est un terme où toutes les variables liées ont une annotation :

$$x^\bullet = x \qquad (M\ N)^\bullet = M^\bullet\ N^\bullet$$
$$(\lambda x : \tau\,.\,M)^\bullet = \lambda x : \tau\,.\,\{[x:\tau]/x\}M^\bullet \qquad [x:\tau]^\bullet = [x:\tau]$$

Notons que même après saturation, il peut exister dans M des variables libres qui n'ont pas d'annotations. Ces variables ne sont pas typables.

Définition 4.10 (Traduction vers le $\lambda_{x\tau}^{\mathsf{st}}$-calcul) Soit $M^\bullet$ un λ_b^{st}-terme saturé. Alors une traduction $M^\bullet$ dans le $\lambda_{x\tau}^{\mathsf{st}}$-calcul est donné par les équations suivantes :

$$[\![\lambda x : \tau\,.\,M]\!] = \lambda x^\tau : \tau\,.\,[\![M]\!]$$
$$[\![M\ N]\!] = [\![M]\!]\,[\![N]\!]$$
$$[\![[x:\tau]]\!] = x^\tau$$

Notons que cette fonction est partielle ; elle n'est pas définie pour les variables sans annotations.

Définition 4.11 (Collection des annotations de type) Notons la concaténation de listes par $(+\!\!+)$. La fonction $\#(\cdot) : \mathsf{Term_b} \to \mathsf{Ctx}$ récolte les annotations de type sur les variables libres dans un terme :

$$\#(x) = [] \qquad \#(M\ N) = \#(M) +\!\!+ \#(N)$$
$$\#(\lambda x : \tau\,.\,M) = \#(M) \setminus x^\tau : \tau \qquad \#([x^\tau : \tau]) = [], x^\tau : \tau$$

L'internalisation des types dans les noms de variables permet de généraliser nombre de propriétés sur les dérivations de typage de la forme $\Gamma \vdash M : A$. En effet, si le contexte de typage Γ est de la forme

$$x_1^{\sigma_1} : \sigma_1, \ldots, x_n^{\sigma_n} : \sigma_n, \tag{4.3}$$

alors il est toujours possible de permuter des variables dans le contexte, de linéariser le contexte ainsi que d'affaiblir un jugement de typage en rajoutant de nouvelles hypothèses n'importe où dans Γ.

Définition 4.12 (Bonne formation d'un contexte, d'un terme) Le contexte de typage vide est bien formé. Inductivement, le contexte $\Gamma,x^\tau : \tau$ est bien formé si Γ est bien formé. La bonne formation de Γ est notée Γ wf.
Lorsque la dérivation de typage descend sous les abstractions, les variables abstraites sont ajoutées au contexte, de sorte qu'il nous faut aussi garantir la bonne formation des termes pour conserver la stabilité de la bonne formation des contextes par extension. Un terme M est bien formé si toutes les abstractions dans M sont de la forme $\lambda x^\sigma : \sigma.\ M$. En particulier, un terme dans l'image de la traduction $[\![\cdot]\!]$ est toujours bien formé.[27]

Dans la suite de cette section, nous utiliserons implicitement les hypothèses de bonne formation sur les termes et sur les contextes pour inférer la forme des noms de variables.

Lemme 4.13 *Pour tout Γ,x,σ,τ tel que Γ wf,*

$$\textit{si } \Gamma \vdash x^\sigma : \tau \textit{ alors } \sigma = \tau.$$

Preuve. Par induction sur Γ. □

Lemme 4.14 (Délayage) *Pour tout contextes bien formés Γ,Δ,Σ, terme bien formé M et types σ,τ,*

1. *si $\Gamma,\Sigma \vdash M : \tau$ alors $\Gamma,x^\sigma : \sigma,\Sigma \vdash M : \tau$;*
2. *si $\Gamma,\Sigma \vdash M : \tau$ alors $\Gamma,\Delta,\Sigma \vdash M : \tau$.*

Preuve. La première partie se démontre par induction sur M. Soit H une dérivation de l'hypothèse

$$\Gamma,\Sigma \vdash M : \tau$$

cas où $M = y$: Par induction sur Σ.

sous-cas où Σ est vide : Si $y = x^\sigma$ alors $\sigma = \tau$ par le lemme 4.13, donc $\Gamma,x^\sigma : \sigma \vdash x^\sigma : \tau$ est toujours prouvable. Sinon, par affaiblissement de H.

[27] Si un terme est bien formé alors l'annotation de type dans les abstractions devient redondante ; le domaine de la fonction que dénote l'abstraction est donné par le nom de la variable abstraite. Nous pourrions donc passer à une variante à la Curry du λ^{st}-calcul où les abstractions ne sont pas annotées — elles sont de la forme $\lambda x^\tau.\ M$. Nous tomberions alors sur la syntaxe des termes de HOL.

sous-cas où $\Sigma = \Sigma', x'^{\sigma'} : \sigma'$: Encore une fois, si $y = x'^{\sigma'}$, alors $\sigma' = \tau$ par le lemme 4.13, donc $\Gamma, x^\sigma : \sigma, \Sigma, x'^{\sigma'} : \sigma' \vdash x'^{\sigma'} : \tau$. Sinon, H se termine forcément par une étape d'affaiblissement, que l'on peut inverser pour obtenir H', une dérivation de $\Gamma, \Sigma \vdash y : \tau$. Puis, par hypothèse d'induction et affaiblissement, $\Gamma, x^\sigma : \sigma, \Sigma, x'^{\sigma'} : \tau \vdash y : \tau$.

cas où $M = \lambda x'^{\sigma'} : \sigma'.\ M'$: Par inversion sur H, τ est de la forme $\sigma' \to \tau'$ et $\Gamma, \Sigma, x'^{\sigma'} : \sigma' \vdash \lambda x' : \sigma'.\ M' : \tau'$. On obtient le résultat par application de l'hypothèse d'induction pour M', puis par la dérivation suivante :

$$(lam)\ \frac{\Gamma, x^\sigma : \sigma, \Sigma, x'^{\sigma'} : \sigma' \vdash M' : \tau}{\Gamma, x^\sigma : \sigma, \Sigma \vdash \lambda x' : \sigma'.\ M' : \tau}$$

cas où $M = M'\ N'$: par hypothèses d'induction.

On démontre la deuxième partie par induction sur Δ, en utilisant le résultat de la première partie. □

Lemme 4.15 (Permutation) *Pour tous contextes bien formés* Γ, Δ*,*

$$\textit{si } \Gamma, x^\sigma : \sigma, x'^{\sigma'} : \sigma', \Delta \vdash M : \tau \textit{ alors } \Gamma, x'^{\sigma'} : \sigma', x^\sigma : \sigma, \Delta \vdash M : \tau.$$

Preuve. Par induction sur la dérivation $\Gamma, x^\sigma : \sigma, x'^{\sigma'} : \sigma', \Delta \vdash M : \tau$. □

Lemme 4.16 (Linéarisation du contexte) *Pour tous contextes bien formés* Γ, Δ, Σ*,*

$$\Gamma, x^\sigma : \sigma, \Delta x^\sigma : \sigma, \Sigma \vdash M : \tau \textit{ si et seulement si } \Gamma, \Delta, x^\sigma : \sigma, \Sigma \vdash M : \tau.$$

Preuve. La démonstration de la première direction se fait par induction sur M. Dans le cas où M est une variable, par induction sur Σ puis Δ, en utilisant le lemme 4.15. L'autre direction est un cas particulier du lemme 4.14. □

Une propriété souhaitable pour la preuve du théorème 4.18 de correction serait que

$$\#(\{[x : \sigma]/x\}N) = \#(N), x^\sigma : \sigma,$$

ou plus généralement que

$$\#(\Gamma, x : \sigma \searrow N) = \#(\Gamma \searrow N), x^\sigma : \sigma.$$

pour tout x, σ, N. Mais cette propriété est seulement vraie modulo permutations d'hypothèses dans les contextes de typage des deux côtés de l'équation. La validité d'un jugement de typage dont le contexte est bien formé, elle, n'est pas sensible aux permutations dans le contexte.

Lemme 4.17 *Pour tout contexte bien formé* Γ*,* λ_b^{st}*-terme* N *et* $\lambda_{x\tau}^{\mathsf{st}}$*-terme* M*, variable* x *et types* σ, τ*,*

$$\textit{si } \#(\Gamma,x : \sigma \searrow N) \vdash M : \tau \textit{ alors } \#(\Gamma \searrow N), x^{\sigma} : \sigma \vdash M : \tau\,.$$

Preuve. Par induction sur M, puis par N dans le cas ou M est une variable. Il nous faudra généraliser l'énoncé du lemme, afin que l'hypothèse d'induction soit suffisamment forte pour qu'une démonstration soit possible dans le cas où N est une application.
Nous cherchons donc à démontrer la propriété plus générale suivante :

$$\text{si } \Delta, \#(\Gamma,x : \sigma \searrow N), \Sigma \vdash M : \tau \text{ alors } \Delta, \#(\Gamma \searrow N), x^{\sigma} : \sigma,\Sigma \vdash M : \tau\,.$$

cas où $M = y^{\tau'}$: Par induction sur N. Soit H la dérivation de $\#(\Gamma,x : \sigma \searrow N)$.

sous-cas où $N = x'$: si $x = x'$, alors par la partie 2 du lemme 4.6 nous avons l'hypothèse $\Delta,x^{\sigma} : \sigma,\Sigma \vdash y : \tau$. Nous procédons par induction sur Σ.
Dans le cas où Σ est vide, nous obtenons le résultat $\Delta, \#(\Gamma \searrow N), x^{\sigma} : \sigma \vdash y : \tau$ par affaiblissement du contexte.
Dans le cas où $\Sigma = \Sigma',x''^{\sigma''} : \sigma''$, alors soit $x''^{\sigma''} = y^{\tau'}$, auquel cas par affaiblissement nous obtenons le résultat, soit $x''^{\sigma''} \neq y^{\tau'}$ et alors le résultat est obtenu par application de l'hypothèse d'induction puis de la règle $(weak)$:

$$\text{(weak)}\ \frac{\Delta, \#(\Gamma \searrow N), \Sigma \vdash y^{\tau} : \tau}{\Delta, \#(\Gamma \searrow N), \Sigma,x''^{\sigma''} : \sigma'' \vdash y^{\tau} : \tau}$$

sous-cas où $N = \lambda x'^{\sigma'} : \sigma'.\ N'$: Pour appliquer l'hypothèse d'induction, il nous faire commuter le déchargement de Γ sous l'abstraction. À la différence du théorème 4.7, où il fallait également faire commuter le déchargement sous l'abstraction, nous ne pouvons pas ici utiliser l'égalité (proposition 4.2), car celle-ci n'est vraie que si le corps de l'abstraction ne contient pas de boîtes. Ici, nous ne pouvons pas garantir l'absence de boîtes. Aussi, nous utilisons plutôt l'identité

$$\Gamma,x : \sigma \searrow \lambda x' : \tau\,.\ M = \lambda x' : \tau\,.\ (\Gamma \setminus x : \tau) \searrow M \tag{4.4}$$

qui se prouve par induction sur Γ. Après réécriture de l'hypothèse H par cette égalité, nous obtenons l'hypothèse

$$\Delta, \#((\Gamma \setminus x' : \sigma') \searrow N'), \Sigma \vdash y^{\tau'} : \tau$$

si $x'^{\sigma'} = y^{\tau'}$,

$$\Delta, \#((\Gamma \setminus x' : \sigma'), x : \sigma \searrow N'), \Sigma \vdash y^{\tau'} : \tau$$

sinon. Nous obtenons le résultat par le lemme 4.14 dans le premier cas et par hypothèse d'induction dans le deuxième.

sous-cas où $N = N_1\ N_2 : \sigma'$: L'hypothèse H est de la forme

$$\Delta,x^{\sigma} : \sigma, (\Gamma,x : \sigma \searrow N_1), (\Gamma,x : \sigma \searrow N_2), \Sigma \vdash y^{\tau'} : \tau\,.$$

Puisque $\Gamma \searrow N_1\ N_2 = (\Gamma \searrow N_1)\ (\Gamma \searrow N_2)$ et par la définition de $\#(\cdot)$, nous cherchons à prouver :

$$\Delta,x^{\sigma} : \sigma,\Gamma \searrow N_1,\Gamma \searrow N_2,\Sigma \vdash y^{\tau'} : \tau$$

alors que nos deux hypothèses d'induction sont

$$\forall\Delta.\forall\Sigma.\forall M\,.\Delta, \#(\Gamma,x : \sigma \searrow N_i), \Sigma \vdash M : \tau \Rightarrow \Delta, \#(\Gamma \searrow N_i), x : \sigma,\Sigma \vdash M : \tau$$

pour $i = 1,2$. Nous obtenons le résultat par application des hypothèses d'induction pour N_1,N_2 et linéarisation du contexte.

sous-cas où N est une annotation : analogue au cas où N est une variable.

cas où M n'est pas une variable : par hypothèse(s) d'induction.

□

Théorème 4.18 (Correction) *Pour tout terme M, contexte de typage Γ et type τ,*

$$\textit{si } \vdash M : \tau \textit{ alors } \#(M) \vdash [\![M^{\bullet}]\!] : \tau\,.$$

Preuve. Par induction sur M. Soit H une dérivation de $\vdash M : \tau$. Dans le cas où M est de la forme $\lambda x : \sigma.\ M'$, par inversion sur H la dérivation sera forcément de la forme

$$(lam)\ \frac{\vdash \{[x : \sigma]/x\}M' : \tau'}{\vdash \lambda x : \sigma.\ M' : \sigma \to \tau'}$$

La prémisse de H porte sur un terme $\{[x : \sigma]/x\}M'$ alors que l'hypothèse d'induction est de la forme $\vdash M' : \tau \Rightarrow \#(M') \vdash [\![M'^{\bullet}]\!]$. Il nous faudrait donc généraliser l'hypothèse d'induction à un préfixe de substitutions appliquées à M' :

$$\begin{array}{l}\vdash (\{[x_1 : \sigma_1]/x_1\}\cdots(\{[x_1 : \sigma_1]/x_1\}M')\cdots) : \tau \\ \quad \Rightarrow \#((\{[x_1 : \sigma_1]/x_1\}\cdots(\{[x_1 : \sigma_1]/x_1\}M')\cdots)) \\ \quad\quad \vdash [\![(\{[x_1 : \sigma_1]/x_1\}\cdots(\{[x_1 : \sigma_1]/x_1\}M')\cdots)^{\bullet}]\!].\end{array}$$

En théorie des types, nous ne pouvons pas raisonner directement par induction sur la taille d'une telle construction, mais nous pouvons raisonner par induction sur une structure de données inductivement définie, par exemple une liste, que nous pouvons traduire vers ce préfixe de substitutions. Nous avons déjà une telle fonction sous la main : c'est précisément le rôle de la

fonction de déchargement que de transformer une liste d'hypothèses de typage en un télescopage de substitutions. Nous généralisons donc l'énoncé du théorème comme suit :

$$\text{pour tout } \Gamma, \text{ si } \vdash \Gamma \searrow M : \tau \text{ alors } \#(\Gamma \searrow M) \vdash [\![(\Gamma \searrow M)^\bullet]\!] : \tau\,.$$

cas où $M = x'$: Par induction sur Γ et en utilisant le lemme 4.6.

cas où $M = \lambda x : \sigma.\ M'$: Par l'équation (4.4), nous avons comme hypothèse $\vdash \lambda x : \sigma.\ (\Gamma \setminus x : \sigma) \searrow M' : \tau$. Par inversion, une dérivation de cette hypothèse est forcément de la forme

$$\frac{\vdash \{[x : \sigma]/x\}((\Gamma \setminus x : \sigma) \searrow M') : \tau'}{\vdash \lambda x : \sigma.\ (\Gamma \setminus x : \sigma) \searrow M' : \sigma \to \tau'}$$

et $\tau = \sigma \to \tau'$. Par définition, $\{[x : \sigma]/x\}((\Gamma \backslash x : \sigma) \searrow M') = (\Gamma \backslash x : \sigma), x^\sigma : \sigma \searrow M'$. On peut donc appliquer l'hypothèse d'induction, puis le lemme 4.17, pour obtenir

$$\#((\Gamma \setminus x : \sigma) \searrow M'), x^\sigma : \sigma \vdash [\![(\Gamma \setminus x : \sigma) \searrow M'^\bullet]\!] : \tau\,.$$

Enfin le résultat est obtenu par la dérivation suivante,

$$(lam)\ \frac{\#((\Gamma \setminus x : \sigma) \searrow M'), x^\sigma : \sigma \vdash [\![(\Gamma \setminus x : \sigma) \searrow M'^\bullet]\!] : \tau}{\#((\Gamma \setminus x : \sigma) \searrow M') \vdash \lambda x^\sigma : \sigma.\ [\![(\Gamma \setminus x : \sigma) \searrow M'^\bullet]\!] : \tau}$$

puis réécriture par l'équation (4.4) en sens inverse, en notant que $\#((\Gamma \setminus x : \sigma) \searrow M') = \#(\lambda x : \tau.\ (\Gamma \setminus x : \sigma) \searrow M')$.

cas où $M = N_1\ N_2 : \sigma'$: par hypothèse d'induction et en utilisant le lemme 4.14.

cas où M **est une annotation** : analogue au cas où M est une variable.

□

Ce théorème de correction ne relie pas M dans le système de type hors contexte avec M dans le système de types standard, parce que ici M est un λ_b^{st}-term sans restrictions. Si nous savons que M ne contient pas d'annotations, et donc que M est aussi un λ^{st}-terme, alors il est possible d'énoncer une propriété de correction reliant M à M.

Lemme 4.19 *Soit la traduction suivante des $\lambda_{x\tau}^{\mathsf{st}}$-termes aux λ^{st}-termes :*

$$\overline{x^\tau} = x \qquad \overline{\lambda x^\tau : \sigma.\ M} = \lambda x : \tau.\ \overline{M} \qquad \overline{M\ N} = \overline{M}\ \overline{N}$$

Pour tout λ_b^{st}-terme M tel que M ne contient pas d'annotations,

$$\overline{[\![M^\bullet]\!]} = M$$

Preuve. Par induction sur M. □

$$(var_{x\tau})\,\frac{}{\vdash^{x\tau} x^{\tau} : \tau} \qquad (abs_{x\tau})\,\frac{\vdash^{x\tau} M : \tau}{\vdash^{x\tau} \lambda x^{\sigma} : \sigma.\, M : \sigma \to \tau}$$

$$(app_{x\tau})\,\frac{\vdash^{x\tau} M : \sigma \to \tau \qquad \vdash^{x\tau} N : \sigma}{\vdash^{x\tau} M\ N : \tau}$$

Figure 4.2 Système de type hors contexte pour les $\lambda^{\mathsf{st}}_{x\tau}$-termes

Corollaire 4.20 *Pour tout terme M tel que M ne contient pas d'annotations, contexte de typage Γ et type τ,*

$$\textit{si } \vdash M : \tau \textit{ alors } [] \vdash M : \tau\,.$$

Preuve. Conséquence du théorème 4.18 et du lemme précédent. Comme M ne contient pas d'annotations, si $\vdash M : \tau$ alors M doit être clos. □

4.2.3 Un algorithme de typage pour les $\lambda^{\mathsf{st}}_{x\tau}$-termes

Si nous devions écrire une fonction pour inférer le type de tout $\lambda^{\mathsf{st}}_{x\tau}$-terme *bien formé*, suivre le schéma du système de type standard serait inefficace puisque le type de toute variable peut être récupéré directement à partir de son nom. Voici donc une fonction (partielle) d'inférence exploitant la forme des noms du $\lambda^{\mathsf{st}}_{x\tau}$-calcul :

$$\begin{aligned} \mathsf{of}(x^{\tau}) &= \tau \\ \mathsf{of}(\lambda x^{\sigma} : \sigma'.\, M) &= \sigma \to \mathsf{of}(M) \\ \mathsf{of}(M\ N) &= \tau \quad \text{si } \sigma = \sigma', \text{ où } \mathsf{of}(M) = \sigma \to \tau \text{ et } \mathsf{of}(N) = \sigma'. \end{aligned}$$

Cet algorithme correspond au système de types de la figure 4.2. Par une simple induction sur M, il est aisé de démontrer que cette fonction est bien définie pour M et $\mathsf{of}(M) = \tau$ si et seulement si $\vdash^{x\tau} M : \tau$. On peut aussi démontrer l'équivalence entre le système de type standard et le système de type de la figure 4.2.

Théorème 4.21 *Pour tout terme M et type τ, $\vdash^{x\tau} M : \tau$ si et seulement si M wf et il existe un contexte de typage Γ tel que Γ wf et $\Gamma \vdash M : \tau$.*

Preuve. La première direction se démontre par induction sur la dérivation de $\vdash^{x\tau} M : \tau$. On utilise le lemme 4.14 d'affaiblissement dans le cas de la règle $(abs_{x\tau})$. Dans le cas de la règle $(app_{x\tau})$, les deux hypothèses d'induction sont

$$\exists \Gamma.\Gamma \text{ wf et } \Gamma \vdash^{x\tau} M : \sigma \to \tau \text{ et } \exists \Gamma'.\Gamma' \text{ wf} \wedge \Gamma' \vdash^{x\tau} N : \sigma,$$

et on cherche à démontrer

$$\exists \Gamma''.\Gamma'' \ \mathsf{wf} \wedge \Gamma'' \vdash^{x\tau} M : \sigma \to \tau \wedge \Gamma'' \vdash^{x\tau} N : \sigma.$$

Γ et Γ' ne sont pas forcément égaux. On ne peut donc instancier Γ'' par Γ, ni par Γ'. Nous l'instancions donc par $\Gamma +\!\!+ \Gamma'$, ce qui permet de prouver le but par délayage des hypothèses d'induction. Puis nous construisons la dérivation suivante :

$$(app_{x\tau})\ \frac{\Gamma,\Gamma' \vdash^{x\tau} M : \sigma \to \tau \qquad \Gamma,\Gamma' \vdash^{x\tau} N : \sigma}{\Gamma,\Gamma' \vdash^{x\tau} M : \tau}$$

La direction inverse se démontre par induction sur la dérivation de $\Gamma \vdash M : \tau$. □

4.3 Du λ-calcul simplement typé aux termes de HOL

Les assistants de preuves exploitant la correspondance de Curry-Howard tendent à unifier le langage des preuves et celui des formules, de sorte que le langage des formules est tout aussi expressif que le langage des preuves, lui-même par nécessité très expressif. D'autres assistants de preuve choisissent au contraire de séparer les preuves et les formules en deux classes syntaxiques distinctes, comme le veut l'usage tout au long de l'Histoire des mathématiques. Les types du λ-calcul simplement typés sont trop pauvres pour permettre d'utiliser les termes de ce calcul pour exprimer des preuves autres que celles de la logique minimale propositionnelle. Mais si l'on choisit un autre langage pour exprimer les preuves dans un assistant, le λ-calcul simplement typé est parfaitement adéquat pour exprimer des formules allant au delà de la simple logique minimale propositionnelle jusqu'à des formules de logiques d'ordres supérieurs, en passant par la logique des prédicats.

Aujourd'hui, les diverses variantes de la famille HOL (HOL4, HOL Light, ProofPower, Isabelle) sont toutes des descendantes de HOL88 (Gordon, 2000), lui-même une variante de LCF (Gordon et al., 1979). Les variantes de HOL héritent de ce dernier l'usage de la théorie des types simples de Church (1940) étendue avec une forme limitée de polymorphisme due à Milner (1978), allié à un système de déduction basé sur la déduction naturelle de Gentzen (1934) plutôt que le système de déduction à la Hilbert (van Heijenoort, 1977) utilisé par Church.

Dans HOL, le type des termes est un type abstrait dont on ne peut construire de valeurs qu'à l'aide de fonctions (formant l'interface de ce type abstrait) qui garantissent que tous les termes sont bien typés au sens de Church, par construction. Par exemple, voici la définition du type des termes dans HOL Light[28] :

```
type term = private
```

[28] http://www.cl.cam.ac.uk/~jrh13/hol-light/

```
    Var of string * hol_type
  | Const of string * hol_type
  | Comb of term * term
  | Abs of term * term
```

La principale particularité de cette définition est que toutes les occurrences de variables sont systématiquement accompagnées du type de celle-ci, ce qui permet de meilleures performances de l'assistant de preuve (paragraphes suivants) et une plus grande simplicité du code (section 4.4).

De nouveaux termes ne peuvent être formés qu'à partir de termes existants à l'aide de l'une des fonctions suivantes :

```
val mk_var : string * hol_type -> term
val mk_const : string * (hol_type * hol_type) list -> term
val mk_abs : term * term -> term
val mk_comb : term * term -> term
```

La fonction `mk_const` vérifie que si le type du deuxième argument est de type σ, alors le type du premier argument est bien de la forme $\sigma \to \tau$. Déterminer le type d'un terme se fait par la fonction `type_of`, dont nous donnons la définition ci-dessous :

```
let rec type_of tm =
  match tm with
    Var(_,ty) -> ty
  | Const(_,ty) -> ty
  | Comb(s,_) -> hd(tl(snd(dest_type(type_of s))))
  | Abs(Var(_,ty),t) -> Tyapp("fun",[ty;type_of t])
```

Puisque la combinaison de deux termes est une opération extrêmement fréquente dans HOL — les termes ne peuvent être formés qu'étape par étape à partir de termes existants — il est impératif qu'un appel à `type_of` soit une opération aussi peu coûteuse en temps calcul que possible. Dans le code ci-dessus, nous voyons que comme les variables (ainsi que les constantes) sont toutes étiquetées par leur type, il n'est pas nécessaire d'allouer de nouvelles cellules sur le tas pour former un environnement que l'on étend dynamiquement au fur et à mesure que l'on descend dans un terme pour déterminer son type, ni de chercher le type d'une variable dans cet environnement.

Revenons à la définition du type `term` des termes. Sachant qu'un terme de la forme $\lambda x^\tau\,.\,M$ est représenté par `Abs(Var(x, tau), M)`, on remarquera que les habitants du type `term` sont une variante à la Curry des $\lambda^{\mathsf{st}}_{x\tau}$-termes introduits dans la section 4.2.2 pour les besoins du

théorème de correction du système de type hors contexte pour le λ-calcul simplement typé. On peut donc interpréter la démonstration du théorème 4.21 comme une formalisation du fait que la fonction `type_of` de HOL est correcte et complète par rapport au système de type standard pour le λ-calcul simplement typé.

4.4 Vers une architecture à la LCF pour les PTS

Un des premiers environnement de formalisation des mathématiques sur ordinateur fut le système AUTOMATH de De Bruijn (1970,1980). Un principe aux motivations philosophiques, pratiques, éducatives, et sociologiques (De Bruijn, 1991) qui a présidé à la conception de AUTOMATH est que la cohérence de l'implémentation du système logique entier doit reposer entièrement sur la propriété de correction d'une entité dont la taille doit être minimisée, que l'on appellera *le noyau*. Qu'un programme informatique soit utile présuppose que l'on fasse confiance à la bonne opération de ce programme. Le principe de De Bruijn affirme la simplicité et la concision de ce programme comme critères nécessaires pour justifier d'une confiance élevée dans la correction des résultats de celui-ci. Comme un assistant de preuves est un artefact complexe par nécessité, une manière de préserver la confiance accordée à l'assistant au fur et mesure de l'ajout de nouvelles fonctionnalités est de s'arranger pour construire le système entier autour d'un noyau dont la taille, elle, est petite et reste grossièrement stable dans le temps.

On retrouve ce principe fondateur dans de nombreux autres assistants, en particulier les descendants de AUTOMATH tels que COQ, AGDA, EPIGRAM, TWELF ou LEGO, au travers de l'élaboration d'un outil dédié vérifiant *a posteriori* les preuves complètes produites par l'utilisateur ou par des tactiques, programmes générateurs de preuves. Mais on retrouve aussi ce principe dans les implémentations dérivées de LCF, contemporain de AUTOMATH. Dans ces derniers systèmes, la cohérence n'est pas externalisée à un vérificateur exécuté dans une phase postérieure. De nouvelles démonstrations ne peuvent être construites que par composition de démonstrations existantes par un ensemble limité de fonctions qui garantissent par construction la bonne formation de toutes les démonstrations en existence. Un vérificateur externe n'est donc pas nécessaire. Dans un système à la LCF, le noyau fondateur est l'ensemble de toutes les fonctions de construction de formules et de démonstrations ainsi que les fonctions attenantes. Seules l'utilisation de cet ensemble restreint de fonctions dites *primitives* relève d'une confiance. Toutes les autres fonctions du système ne peuvent synthétiser de nouvelles preuves qu'à l'aide de ces primitives — le résultat de ces fonctions est donc forcément correct si les primitives le sont.

Dans HOL LIGHT, les fonctions de construction des termes et la fonction `type_of` présentées ci-dessus forment une partie de l'interface du noyau fondamental de ce système. Les autres

fonctions du noyau servent essentiellement à déclarer de nouveaux axiomes, à introduire de nouvelles définitions et à construire de nouvelles démonstrations.

L'objectif de ce chapitre est d'élaborer une présentation des PTS qui soit compatible avec une architecture à la LCF. Les assistants de preuves descendants de AUTOMATH cités plus haut sont tous basés sur diverses extensions ou variations d'instances particulières du formalisme des PTS, tels que le Edinburgh Logical Framework (Avron et al., 1989), le Calcul des Constructions (Coquand et Huet, 1988), étendu ou non avec des types inductifs (Coquand et Paulin, 1990 et Werner, 1994) ou avec des univers (Coquand, 1986), ou encore la théorie des types de Martin-Löf (1984). Une version compatible des PTS serait une étape vers des implémentations alternatives de ces assistants selon les mêmes principes de conception que LCF, unifiant de la sorte deux des principales lignées historiques d'assistants de preuves.

L'intérêt d'une telle démarche est double : la simplification du noyau, selon les grandes lignes de la section 4.1, et la performance de l'assistant.

4.4.1 Le choix du métalangage

Tous les assistants dans la lignée de AUTOMATH proposent un métalangage dans lequel exprimer les démonstrations et définir diverses notations et abréviations facilitant la lecture de celles-ci. Dans COQ, le métalangage GALLINA est composé d'un sous-langage $\mathcal{L}_{tac}$ (Delahaye, 2000 et Delahaye, 2002) dédié à l'écriture de tactiques. Ce langage est un véritable langage de programmation d'une étonnante versatilité, suffisamment puissant pour écrire en $\mathcal{L}_{tac}$ de nombreux programmes, petits ou grands, qui construisent des démonstrations. Ces programmes, appelés *tactiques*, peuvent être composés pour créer de nouveaux programmes plus puissant encore.

Ces dernières années ont vu une exploitation accrue du métalangage pour une automatisation de plus en plus poussée du processus de démonstration, que ce soit des tactiques pour résoudre des domaines spécialisés (Chlipala et al., 2009), ou des tactiques généralistes (Chlipala, 2010a et Chlipala, 2010b). Par exemple, le projet Ynot (Nanevski et al., 2008a) utilise $\mathcal{L}_{tac}$ pour implémenter une série de procédures pour simplifier voir résoudre les obligations de preuves émanant des conditions de bord de la logique de séparation[29] utilisée pour exprimer et démontrer des propriétés sur des programmes impératifs manipulant des pointeurs.

Traditionnellement, les tactiques les plus complexes de COQ, telles que la procédure de décision Omega (Pugh, 1991 et Crégut, 2004) ou encore la tactique de résolution des sous-but triviaux `auto` n'étaient pas implémentées dans $\mathcal{L}_{tac}$, mais plutôt dans le langage OCAML qui sert de langage d'implémentation de COQ, pour des raisons historiques mais aussi et surtout pour des raisons de performance. $\mathcal{L}_{tac}$ est pour l'heure un langage interprété pour lequel il n'existe

[29] Une extension de la logique de Hoare (1969).

pas de compilateur vers du bytecode ou du code natif. Si un nombre croissant de tactiques est amené à être implémenté en $\mathcal{L}_{tac}$, alors il faudra bien développer une plate-forme d'exécution plus efficace pour ce langage.

Peu importe le langage d'implémentation des tactiques, OCaml ou $\mathcal{L}_{tac}$, ni l'un ni l'autre n'est le langage des termes de preuves de la logique de Coq, alors que lui aussi repose sur un environnement d'exécution dédié et sophistiqué pour assurer une performance raisonnable de la vérification d'une large classe de preuves appelées preuves réflexives.

4.4.2 Les preuves par réflexion

Un style de preuves aux applications grandissantes consiste exploiter les capacités calculatoires du langage des termes et des preuves afin de remplacer nombre d'étapes déductives par un calcul. Ce sont les preuves par réflexion. Celles-ci échangent un rétrécissement de la taille des preuves pour un temps de calcul plus long de la part du noyau de vérification. Dans la pratique, ce temps de calcul plus long est souvent largement compensé par un temps global de vérification plus court car les termes de preuves sont plus petits.

Pour un prédicat P portant sur des termes de type T, l'idée est d'introduire une procédure de décision $\mathsf{f} : T \Rightarrow \mathsf{bool}$ et sa preuve de correction $\mathsf{f}_{\mathsf{correct}} : \forall x : T\,.\,\mathsf{f}\ x = \mathsf{true} \Rightarrow P\ x$. Ainsi, plutôt que de démontrer directement une propriété de la forme

$$\forall t : T\,.\,P\ t,$$

on peut fournir le terme $\lambda t : T\,.\ \mathsf{f}_{\mathsf{correct}}\ t\ (\mathsf{refl\ true}) : \forall t : T\,.\ P\ t$ comme preuve de cette propriété, où $\mathsf{refl\ true}$ est la preuve canonique de $\mathsf{true} = \mathsf{true}$.

Par exemple, considérons le prédicat le défini comme suit :

$$\mathsf{le} : \mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathbf{Type}$$
$$\mathsf{le}_{\mathsf{n}} : \forall n : \mathsf{nat}.\ \mathsf{le}\ n\ n$$
$$\mathsf{le}_{\mathsf{S}} : \forall n, m : \mathsf{nat}.\ \mathsf{le}\ n\ m \Rightarrow \mathsf{le}\ n\ (S\ m)$$

On pourrait aussi décider l'ordre $(\leq)$ à l'aide de la fonction suivante :

$$\mathsf{leb} : \mathsf{nat} \Rightarrow \mathsf{bool}$$
$$\mathsf{leb}\ n\ m = \mathsf{true} \qquad \text{si } n - m = 0$$
$$\mathsf{leb}\ n\ m = \mathsf{false} \qquad \text{sinon}$$

Pour deux entiers n,m donnés tel que $n \leq m$, $\mathsf{f}_{\mathsf{correct}}$ nous dit que comme toute preuve de $\mathsf{leb\ n\ m} = \mathsf{true}$ peut être transformée en une preuve de $\mathsf{le}\ n\ m$, il suffit de démontrer

leb n m $=$ true, dont une preuve particulièrement succincte est refl true, de taille constante pour tous n,m. La taille de cette preuve est à contraster avec celles de le n m, dont même les formes normales sont proportionnelles à la différence entre n et m. On a ainsi remplacé une démonstration de le n m (à l'aide des constructeur le$_0$, le$_S$) par un calcul de $n - m$.

La bibliothèque de théories combinatoires de base (arithmétique, listes, ensembles finis) Ssreflect (Gonthier et Mahboubi, 2008) utilise systématiquement ce style de preuves pour réduire la taille des preuves, simplifier l'effort de démonstration pour l'utilisateur dans le cadre de théories décidables et faciliter le raisonnement par réécriture. La méthodologie Ssreflect est à la base de la preuve du théorème des quatre couleurs de Gonthier (2007), où le style réflexif est utilisé à grande échelle. Il existe également un corpus grandissant de procédures de décision implémentées dans Coq dans ce style (Crégut, 2004, Grégoire et Mahboubi, 2005, Besson, 2006, Lescuyer et Conchon, 2008 et Braibant et Pous, 2010). Que la réflexion soit utilisée à petite échelle de nombreuses fois ou peu de fois mais induisant de gros calculs (comme dans le théorème des quatre couleurs ou certaines procédures de décisions (Joshi et Mahboubi, 2009)), la performance de la réduction des termes est cruciales pour la viabilité de cette méthodologie en pratique.

Malgré l'unification du langage de formules, des termes et des démonstrations, on recense donc au moins trois langages dans un assistant de preuves complet basé sur la théorie des types : le langage de l'implémentation, le langage des termes et le langage des tactiques. Nous avons vu que la performance est cruciale pour chacun de ces trois langages. Il serait souhaitable que ces trois langages puissent donc partager un maximum des technologies d'exécution efficaces mises en oeuvre pour l'un ou l'autre de ces langages, afin de brider la complexité des assistants et libérer du temps de développement des implémenteurs pour d'autres questions plus intéressantes que la simple performance. La réponse radicale qu'offre LCF est d'unifier deux de ces trois langages, d'implémentation et de tactiques, quitte à introduire du sucre syntaxique spécifique pour chacun des trois domaines : implémentation, termes, tactiques. Une architecture dans le style de LCF pour les PTS est une première étape vers un assistant de preuves basé sur la théorie des types qui unifie le langage de l'implémentation et des tactiques. Nous verrons dans une étape ultérieure comment aller plus loin que LCF et unifier ces deux langages avec celui des termes, ce qui permettra de réutiliser les mécanismes du langage d'implémentation pour le test de convertibilité de deux termes (absent de HOL et de la logique des fonctions calculables de Scott (1993) à la base de LCF).

4.4.3 Vers des PTS hors contexte

Imaginons un noyau à la LCF pour un PTS arbitraire $\mathcal{P}$. Une des fonctions formant l'interface de noyau serait `mkApp`, dont le rôle serait analogue à celui de la fonction de HOL LIGHT du même nom. Cette fonction aurait comme signature (en HASKELL)

```
mkApp :: (Context, Term) -> (Context, Term) -> (Context, Term)
```

où les arguments à `mkApp` sont des termes potentiellement ouverts dont le type de chacune des variables libres est donné par des contextes adjacents. Former l'application du terme M clos par Γ au terme N clos par Δ donnerait un nouveau terme $M\ N$ clos par $\Gamma +\!\!+ \Delta$. Mais ce terme n'existe que si les contextes Γ et Δ sont compatibles, c'est à dire que pour tout $x : A \in \Gamma$ et $x : B \in \Delta$ alors $A = B$. Par exemple on ne peut construire d'application $x\ x$ bien typée à partir des clôtures $(x : A \to B,x)$ et $(x : A,x)$.

La fonction `mkApp` devra donc vérifier la compatibilité des contextes à chaque formation d'une application. Sachant qu'en pratique un terme peut contenir des centaines de variables libres, tester la compatibilité de deux gros contextes à chaque noeud d'application serait prohibitif. C'est sans doute cette raison qui a conduit la plupart des assistants de preuve à base de PTS à délaisser l'architecture LCF[30] au profit de l'architecture de AUTOMATH.

Nous généralisons donc dans les prochaines sections la théorie des types simples hors contexte développée plus tôt à tous les PTS, afin d'ouvrir l'architecture LCF au delà de HOL et ses types simples à des théories avec types dépendants.

4.5 Les PTS hors contexte

Définition 4.22 (λ_b^{pts}-calcul) Soit un PTS $\langle \mathcal{S},\mathcal{A},\mathcal{R}\rangle$. Les termes du λ_b^{pts}-calcul sont caractérisés inductivement par la grammaire en forme Backus-Naur suivante :

$$\begin{array}{rcll} \mathcal{S} & \ni & s, s_n & \\ \mathsf{Var} & \ni & x, y, z & \\ \mathsf{Term_b} & \ni & M, N, A, B \quad ::= \quad s \mid x \mid \lambda x : A.\ M \mid \Pi x : A.\ B \mid M\ N \mid [x : A] & \end{array}$$

L'opération de substitution d'un terme pour une variable substitue aussi dans le type des boîtes. Nous étendons les égalité définissant la substitution dans la définition 1.14 avec les égalités suivantes (ou l'on suppose $x \neq y$) :

[30] Une exception notable est le système LEGO de Pollack (Luo et Pollack, 1992).

$$(cast_b)\ \frac{\vdash A : s}{\vdash [x : A] : A} \qquad (sort_b)\ \frac{}{\vdash s_1 : s_2}\ \text{si } \langle s_1,s_2\rangle \in \mathcal{A}$$

$$(abs_b)\ \frac{\vdash \{[x : A]/x\}M : B \qquad \vdash \Pi x : A.\ B : s}{\vdash \lambda x : A.\ M : \Pi x : A.\ B}$$

$$(prod_b)\ \frac{\vdash A : s_1 \qquad \vdash \{[x : A]/x\}B : s_2}{\vdash \Pi x : A.\ B : s_3}\ \text{si } \langle s_1,s_2,s_3\rangle \in \mathcal{R}$$

$$(app_b)\ \frac{\vdash M : \Pi x : A.\ B \qquad \vdash N : A}{\vdash M\ N : \{N/x\}B} \qquad (conv_b)\ \frac{\vdash M : A \qquad \vdash B : s}{\vdash M : B}\ \text{si } A =_\beta B$$

Figure 4.3 Système de type hors contexte pour un PTS donné par un tuple $\langle \mathcal{S},\mathcal{A},\mathcal{R}\rangle$, l'ensemble des sortes, l'ensemble des axiomes et l'ensemble des règles.

$$\{N/x\}[x : A] = N$$
$$\{N/x\}[y : A] = [y : \{N/x\}A]$$

Les notions de clôture par contexte d'une relation de réduction et de α-conversion sont étendues de la manière usuelle aux termes du λ_b^{pts}-calcul. La β-réduction est la même que pour le λ-calcul.

La figure 4.3 présente les règles d'un système de types pour les λ_b^{pts}-termes. Comme dans le cas du λ-calcul simplement typé, les variables ne sont pas typables, mais les annotations le sont.

4.5.1 Variables annotées ou noms typés

Une alternative aurait consisté à définir la substitution à l'aide des égalités de la définition 1.14 étendues avec les égalités suivantes (où l'on suppose $x \neq y$) :

$$\{N/x\}[x : A] = N$$
$$\{N/x\}[y : A] = [y : A]$$

Ne plus substituer dans le type des boîtes revient à considérer que $[x : A]$ tout entier est un nom de variable, au même titre que x. On pourrait alors dire qu'il existe deux classes de noms dans nos termes : les noms habituels et les noms typés. C'est d'ailleurs l'approche adoptée par Geuvers et al. (2010). Dans leur système Γ_∞ présenté, notre $[x : A]$ est noté x^A.

Dans le cas du λ-calcul simplement typé, ce changement n'aurait aucun impact. Mais Geuvers et al. (2010) remarquent que la possibilité de types dépendants introduit une nouvelle subtilité par rapport au système de types hors contexte pour le λ-calcul simplement typé introduit dans la section 4.2. Dans le système de types standard pour les PTS, l'ordre des hypothèses dans le contexte de typage n'est pas arbitraire. Même si le contexte est linéaire, les

hypothèses du contexte ne peuvent être permutée arbitrairement, comme c'est le cas pour le λ-calcul simplement typé. En effet, le type d'une variable peut dépendre d'hypothèses antérieures dans le contexte. L'ordre des hypothèses dans le contexte est d'autant plus important qu'il permet de restreindre la construction des abstractions de manière à éviter de compromettre la cohérence du PTS comme logique.

Supposons deux sortes $*, \Box$ avec $* : \Box$, un ensemble de constantes

$$A : *, P : A \to *, Q : \Pi x : A.\ P\ x \to *, a : A, h : P\ a$$

et considérons la dérivation partielle suivante tirée d'un exemple dû à Geuvers et al. (2010) :

$$\frac{\vdash Q^{\Pi x:A^*.\ P^{A^*\to *}\ x\to *}\ a^{A^*}\ h^{P^{A^*\to *}\ a^{A^*}} : *}{\vdash \Pi a : A^*.\ Q^{\Pi x:A^*.\ P^{A^*\to *}\ x\to *}\ a\ h^{P^{A^*\to *}\ a^{A^*}} : *}$$

Cette dérivation montre que si la substitution ne descend pas dans le type des annotations alors le système de types de la figure 4.3 devient incohérent, puisque dans la conclusion h n'est pas de type $P\ y$. Cette dérivation correspondrait à la dérivation suivante dans le système de types standard pour les PTS si celle-ci était légale :

$$\frac{A : *, P : A \to *, Q : \Pi x : A.\ P\ x \to *, a : A, h : P\ a \vdash Q\ a\ h : *}{A : *, P : A \to *, Q : \Pi x : A.\ P\ x \to *, h : P\ a \vdash \Pi y : A.\ Q\ y\ h : *}$$

Il devient apparent que la dérivation construit un produit dépendant par abstraction d'une variable dont l'hypothèse de typage $y : A$ dans le contexte n'est pas en position terminale. Cette hypothèse ne peut être permutée en position terminale car le type de h dépend de y. Là où abstraire une variable est un acte régi par la forme du contexte dans le système de type standard, ici l'absence de contexte ouvre la porte à toutes sortes d'abstractions abusives.

Geuvers et al. introduisent une notion *d'ensemble héréditaire des variables libres des types des variables libres* hfvT, c'est à dire l'union des ensembles de variables libres du type qui sert d'étiquette à chaque nom de variable typé apparaissant dans le terme. Le problème de cohérence est résolu en imposant que l'abstraction d'une variable x dans un terme M ne se fasse que si $x \notin \mathsf{hfvT}(M)$. Cette restriction n'est pas nécessaire ici mais la théorie que nous élaborons, ici et dans le prochaines sections, en devient très différente. En particulier,

$$x^{(\lambda A^*.\ A)\ B^*} \not\equiv_\beta x^{B^*},$$

dans Γ_∞, alors que pour les λ_b^{pts}-termes présentés ici nous avons

$$[x : (\lambda A : *.\ A)\ [B : *]] \equiv_\beta [x : [B : *]].$$

4.5.2 Variables ordonnées

Dans le système de types standard pour les PTS de la section 2.3, il n'est possible d'abstraire une variable x que si aucune autre hypothèse du contexte ne dépend de x. Cette restriction est implémentée dans les règles de typage en imposant que les hypothèses dans le contexte sont toujours en ordre topologique. Comme l'abstraction ne peut se faire que sur la variable dont l'hypothèse de typage est en dernière position dans le contexte, celle-ci est forcément une racine du graphe des dépendances associé au contexte. Aucune autre hypothèse ne dépend donc de cette variable.

Prenons le terme $M = [x : [y : *]]$. Ce terme correspond à la variable x sous contexte $\Gamma = y : *, x : y$. Dans le système de la figure 4.3, il est possible d'abstraire sur n'importe laquelle des variables x,y :

$$\frac{\ldots \qquad \vdash [x : [y : *]] : [y : *]}{\vdash \Pi x : [y : *].\ x : *} \quad \text{ou} \quad \frac{\ldots \qquad \vdash [x : [y : *]] : [y : *]}{\vdash \Pi y : *.\ [x : y] : *}$$

alors que le système de types standard ne permet que la dérivation

$$\frac{\ldots \qquad y : *, x : y \vdash x : *}{y : * \vdash \Pi x : y.\ x : *}$$

Le schéma de systèmes de la figure 4.3 est donc plus permissif que les PTS standards. Établir la cohérence logique de systèmes instances de ce schéma en devient une tâche plus ardue. Cette propriété reste une conjecture dont la résolution est laissée à de futurs travaux[31]. Nous préférons imposer dans cette section une restriction sur l'ordre des abstractions afin que les PTS hors contextes héritent des bonnes propriétés de leur pendant contextuel. Nous montrerons dans la section 4.6 comment cette restriction est une conséquence naturelle d'une stratégie de représentation des termes pratique et efficace.

Nous distinguons deux classes de noms, les noms de variables libres et les noms de variables liées, les premiers étant par simplicité des entiers naturels. Nous aurions pu tout aussi bien choisir un autre ensemble de noms muni d'un ordre total sur ses éléments pour les variables libres.

Définition 4.23 ($\lambda_{bn}^{\mathsf{pts}}$-calcul) Soit un PTS $\mathcal{P} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$. La syntaxe des $\lambda_{bn}^{\mathsf{pts}}$-termes est donné par la grammaire suivante :

[31] Si nous considérons le terme $\Pi y : *.\ [x : y]$ dans une logique intuitionniste, il y a clairement extrusion de la portée de la variable y, puisque y est le type de x à l'intérieur comme à l'extérieur de la portée introduite par le produit dépendant. Mais une piste serait de considérer x comme une variable modale et le produit dépendant de ce terme comme un opérateur modal d'une théorie des types modale (Nanevski et al., 2008b).

$$(cast_{bn})\ \frac{\vdash_m A : s}{\vdash_n [m : A] : A}\ m < n \qquad (sort_{bn})\ \frac{}{\vdash_n s_1 : s_2}\ \text{si } \langle s_1,s_2\rangle \in \mathcal{A}$$

$$(abs_{bn})\ \frac{\vdash_{n+1} \{[n : A]/x\}M : \{[n : A]/x\}B \qquad \vdash_n \Pi x : A.\ B : s}{\vdash_n \lambda x : A.\ M : \Pi x : A.\ B}$$

$$(prod_{bn})\ \frac{\vdash_n A : s_1 \qquad \vdash_{n+1} \{[n : A]/x\}B : s_2}{\vdash_n \Pi x : A.\ B : s_3}\ \text{si } \langle s_1,s_2,s_3\rangle \in \mathcal{R}$$

$$(app_{bn})\ \frac{\vdash_n M : \Pi x : A.\ B \qquad \vdash_n N : A}{\vdash_n M\ N : \{N/x\}B} \qquad (conv_{bn})\ \frac{\vdash_n M : A \qquad \vdash_n B : s}{\vdash_n M : B}\ \text{si } A =_\beta B$$

Figure 4.4 Système de type hors contexte pour un PTS donné par un tuple $\langle \mathcal{S},\mathcal{A},\mathcal{R}\rangle$, l'ensemble des sortes, l'ensemble des axiomes et l'ensemble des règles.

$$\begin{array}{rcll} \mathsf{Var} & \ni & x, y, z & \\ \mathsf{Parameter} & \ni & k, n, m & = \quad \mathbb{N} \\ \mathsf{Term_{bn}} & \ni & M, N, A, B & ::= \quad s \mid x \mid n \mid \lambda x : A.\ M \mid \\ & & & \qquad \Pi x : A.\ B \mid M\ N \mid [n : A] \end{array}$$

Cette distinction est classique dans certaines présentations de la logique (Paulson, 1992 et Prawitz, 1965) où l'on distingue *paramètres* (variables libres) et variables (liées). Nous la retrouvons également dans certaines formalisations de la théorie de types (Coquand, 1991, McKinna et Pollack, 1993 et Pollack, 1994). La figure 4.4 donne les règles d'un système de types dont les jugements sont paramétrés par un entier n jouant le rôle de compteur de variables libres dans le terme et son type.

Lemme 4.24 *Pour tous termes M,A, si $\vdash_n M : A$ alors $\forall m,m \in \mathcal{FV}(M) \cup \mathcal{FV}(A) \Rightarrow m < n$.*

Preuve. Par induction sur la dérivation de $\vdash_n M : A$. □

La règle de typage intéressante est celle donnant le type des boîtes. La condition de bord de cette règle vérifie que le paramètre n des dérivations est une borne supérieure des variables libres dans le terme, de sorte qu'il puisse être utilisé comme générateur de noms frais pour nommer une variable que l'on libère (comme dans le typage des abstractions et des produits) sans risque de captures. De plus, une boîte est typable si et seulement si toutes les variables libres du type de la boîte sont inférieures à la variable m typée par la boîte. Cette condition exclut en particulier toute dérivation de typage du terme $\Pi y : *.\ [x : y]$, comme l'illustre la tentative de dérivation suivante :

$$\dfrac{\vdash_0 * : \Box \qquad \dfrac{\dfrac{\vdash_0 * : \Box}{\vdash_0 [1 : *]}\, 1 < 0}{\vdash_2 [0 : [1 : *]]}\, 0 < 2}{\vdash_1 \Pi y : *.\ [0 : y]}$$

Le lieur de x précède celui de y donc le numéro de x est choisi tel qu'il soit inférieur au numéro de y. Dans cette dérivation, $1 \not< 0$ et donc la condition de bord d'une des applications de $(cast_{bn})$ n'est pas respectée.

4.6 PTS hors contexte en HOAS

Les systèmes des sections précédentes ignorent les problèmes d'α-conversion et le coût des substitutions lors du typage des applications. Un traitement formel des PTS hors contexte se doit d'être précis sur ces points, à la différence d'un traitement informel sur papier, où l'on se contente souvent d'arguments *ad hoc* telle que l'usage de la convention de Barendregt sans la justifier. Convaincre un assistant de preuves du bien fondé de ses énoncés est plus difficile. Il existe quantité de styles de formalisation de termes avec lieurs, que nous ne pouvons tous citer ici. Aydemir et al. (2005) donnent une analyse relativement complète des différentes approches développées dans la littérature.

Nous choisissons dans cette section de présenter les PTS hors contexte avec des termes encodés avec de la syntaxe abstraite d'ordre supérieur. Cette approche nous semble particulièrement naturelle ici car on y retrouve la même distinction entre variables libres et variables liées que dans la section précédente. Cet encodage des termes délègue le problème de la substitution au métalangage et de ce fait admet une implémentation relativement efficace. Nous pouvons donc réutiliser ce même encodage des termes dans l'implémentation de l'algorithme de typage dans un noyau de vérification de preuves réel tel que Dedukti. La formalisation dans un assistant de preuves des PTS hors contexte et de l'algorithme de vérification associé donne directement une formalisation de l'implémentation de Dedukti.

Définition 4.25 ($\lambda^{\mathsf{pts}}_{bh}$-calcul) Soit un PTS $\mathcal{P} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$. La syntaxe des $\lambda^{\mathsf{pts}}_{bh}$-termes est donnée par la grammaire suivante :

$$\begin{array}{rcccl} \mathsf{Fun} & \ni & f, g & ::= & \overline{\lambda} x.\ M \\ \mathsf{Term_{bh}} & \ni & M, N, A, B & ::= & s \mid x \mid \mathsf{F}\ n \mid \mathsf{Lam}\ A\ f \mid \\ & & & & \mathsf{Pi}\ A\ f \mid \mathsf{App}\ M\ N \mid [n : A] \end{array}$$

La syntaxe du $\lambda^{\mathsf{pts}}_{bh}$-calcul est un λ-calcul à deux niveaux. Les abstractions de ce langage sont un type allié à une fonction du métalangage (dont le lieur est toujours surligné) des $\lambda^{\mathsf{pts}}_{bh}$-termes

vers les $\lambda_{bh}^{\mathsf{pts}}$-termes. Le produit dépendant est identique. Par abus, nous réutiliserons parfois les métavariables M ,N ,A ou B pour désigner des éléments syntaxiques de la classe Fun des fonctions du métalangage sur les termes.

On reconnaîtra le $\lambda_{bh}^{\mathsf{pts}}$-calcul comme l'image d'un schéma de représentation de Mogensen, mais pour un calcul typé. La fonction d'encodage des $\lambda_{bn}^{\mathsf{pts}}$-termes vers les $\lambda_{bh}^{\mathsf{pts}}$-termes, ou schéma de représentation, est donnée comme suit :

$$\ulcorner s \urcorner = s \qquad \ulcorner x \urcorner = x \qquad \ulcorner n \urcorner = \mathsf{F}\ n$$
$$\ulcorner \lambda x : A.\ M \urcorner = \mathsf{Lam}\ \ulcorner A \urcorner\ (\overline{\lambda} x.\ \ulcorner M \urcorner) \qquad \ulcorner \Pi x : A.\ B \urcorner = \mathsf{Pi}\ \ulcorner A \urcorner\ (\overline{\lambda} x.\ \ulcorner B \urcorner)$$
$$\ulcorner M\ N \urcorner = \mathsf{App}\ \ulcorner M \urcorner\ \ulcorner N \urcorner \qquad \ulcorner [n : A] \urcorner = [n : \ulcorner A \urcorner]$$

Ce schéma de représentation diffère de celui donné à la page 66 sur un point de détail : les variables liées ne sont pas représentées à l'aide d'un constructeur B. Cette différence correspond à la différence entre HOAS et WHOAS, c'est à dire la syntaxe abstraite d'ordre supérieure faible (Despeyroux et al., 1995 et Miculan, 2001). Un désavantage du WHOAS est que la substitution devient plus complexe. Aussi nous préférons présenter ici une version du $\lambda_{bh}^{\mathsf{pts}}$-calcul en HOAS, qui est celle que nous avons choisi dans l'implémentation de Dedukti. Nous reviendrons cependant sur le WHOAS dans la section 4.6.1.

Par souci de concision, nous introduisons les synonymes

$$\mathsf{Lam} = \lambda \qquad \mathsf{Pi} = \Pi \qquad \mathsf{App} = \alpha,$$

et écrivons simplement m au lieu de $\mathsf{F}\ m$. La figure 4.5 montre un système de types hors contexte directement formalisable dans un assistant de preuves basé sur la théorie des types tel que Coq. Ici, la substitution est encodée comme une simple application du métalangage, ce qui évite d'avoir à implémenter puis à raisonner sur une fonction de substitution telle que donnée dans le chapitre 1 et les délicats renommages de variables associés.

La règle de conversion dans la figure 4.5 fait appel à une relation $(\equiv_\beta^n)$ paramétrée par un entier n, comme les règles de typage. Cette relation de conversion est définie comme suit.

Définition 4.26 (Conversion sur les $\lambda_{bh}^{\mathsf{pts}}$-termes) Soit deux $\lambda_{bh}^{\mathsf{pts}}$-termes M ,M'. Nous disons que M et M' sont *n-convertibles* (ou simplement *convertibles*) si ils sont reliés par la clôture reflexive, symétrique et transitive de la relation $(\rightrightarrows_\beta^n)$, que nous notons $M \equiv_\beta^n M'$.

Nous démontrons la complétude dans la section 4.6.2 et la correction dans la section 4.6.3 de ce système de type hors contexte par rapport au système de type standard pour les PTS du chapitre 2.

$$(cast_{hn})\ \frac{\vdash_m A : s}{\vdash_n [m : A] : A}\ m < n \qquad\qquad (sort_{hn})\ \frac{}{\vdash_n s_1 : s_2}\ \text{si } \langle s_1,s_2\rangle \in \mathcal{A}$$

$$(abs_{hn})\ \frac{\vdash_{n+1} M\ [n : A] : B\ [n : A] \qquad \vdash_n \Pi\ A\ B : s}{\vdash_n \lambda\ A\ M\ : \Pi\ A\ B}$$

$$(prod_{hn})\ \frac{\vdash_n A : s_1 \qquad \vdash_{n+1} M\ [n : A] : s_2}{\vdash_n \Pi\ A\ M\ : s_3}\ \text{si } \langle s_1,s_2,s_3\rangle \in \mathcal{R}$$

$$(app_{hn})\ \frac{\vdash_n M\ : \Pi\ A\ B \qquad \vdash_n N\ : A}{\vdash_n M \cdot N\ : B\ N}$$

$$(conv_{hn})\ \frac{\vdash_n M\ : A \qquad \vdash_n B : s}{\vdash_n M\ : B}\ \text{si } A \equiv^n_\beta B$$

Figure 4.5 Système de type hors contexte pour un PTS donné par un tuple $\langle \mathcal{S},\mathcal{A},\mathcal{R}\rangle$, l'ensemble des sortes, l'ensemble des axiomes et l'ensemble des règles.

$$\frac{}{s \rightrightarrows^n_\beta s} \qquad \frac{}{x \rightrightarrows^n_\beta x}$$

$$\frac{M\ n \rightrightarrows^{n+1}_\beta M'\ n \qquad N \rightrightarrows^n_\beta N'}{(\lambda\ A\ M)\ N \rightrightarrows^n_\beta M'\ N'}$$

$$\frac{A \rightrightarrows^n_\beta A' \qquad M\ n \rightrightarrows^{n+1}_\beta M'\ n}{\lambda\ A\ M \rightrightarrows^n_\beta \lambda\ A'\ M'} \qquad \frac{A \rightrightarrows^n_\beta A' \qquad B\ n \rightrightarrows^{n+1}_\beta B'\ n}{\Pi\ A\ B \rightrightarrows^n_\beta \Pi\ A'\ B'}$$

$$\frac{M \rightrightarrows^n_\beta M' \qquad N \rightrightarrows^n_\beta N'}{M \cdot N \rightrightarrows^n_\beta M' \cdot N'}$$

Figure 4.6 Relation de réductions parallèles sur les termes encodés en HOAS.

4.6.1 Formalisation dans Coq

Dans les sections suivantes, nous supposons que le métalangage est le calcul des construction inductives avec Set prédicatif. Nous postulons un seul axiome, compatible avec cette théorie.

Axiome 4.1 (Extensionnalité fonctionnelle) *Soit A,B deux types. Pour tous f ,$g : A \to B$,*

$$(\forall x.\ f\ x = g\ x) \Rightarrow f\ = g.$$

Le problème principal d'un encodage à l'ordre supérieur de termes dans un tel métalangage est d'éviter les termes exotiques. L'expressivité du calcul des constructions inductives rend son usage comme métalangage incompatible avec un encodage en HOAS du langage objet. En effet imaginons que Coq accepte la définition de type inductif suivant :

```
Inductive tm :=
| λ : (tm -> tm) -> tm
| app : tm -> tm -> tm.
```

Il devient alors possible de définir le terme suivant :

```
Definition exotic := λ (fun x => match x with
     | λ f => f x
     | app M N => app M N end).
```

Ce terme ne correspond à aucun terme du λ-calcul puisque le corps de la fonction dépend de l'argument de la fonction. Pire encore : la fonction dans `exotic` appliquée au terme `exotic` est un terme Coq sans forme normale. L'existence d'un tel terme compromet la décidabilité du typage et la cohérence logique de Coq. Pour cette raison, le CIC distingue les définitions strictement positives de celle qui ne le sont pas et interdit ces dernières. Le type inductif `tm` n'est pas strictement positif : nous remarquons une occurrence négative de `tm` dans le type du constructeur `λ`.

Une méthode d'encodage particulièrement attractive dans Coq consiste à maintenir abstrait le type des représentations des valeurs des variables. On obtient ainsi une syntaxe abstraite d'ordre supérieur paramétrique (PHOAS) (Chlipala, 2008), faisant d'une pierre deux coups : le type inductif encodant un langage objet en PHOAS est strictement positif et les termes exotiques tel que `exotic` sont exclus car il devient impossible de discriminer sur la représentation d'une variable dans le corps d'une fonction. En PHOAS, le type des λ-termes purs est donné par

```
Inductive tm (v : Set) :=
| λ : (v -> tm v) -> tm v
| α : tm v -> tm v -> tm v.
```

On peut ensuite instancier `v` par `tm` lui-même :

```
Definition tm' := tm tm.
```

ce qui nous permet d'écrire une opération de substitution

```
Definition subst N M := flatten (M N).
```

en termes d'une fonction `flatten` donné comme suit :

```
Fixpoint flatten v (M : tm (tm v)) : tm v :=
match M with
  | λ M => λ (fun x => flatten (M (β x)))
  | α M N => α (flatten M) (flatten N)
end.
```

L'aplatissement et le rehaussement intempestif de termes dû à l'encodage peut devenir gênant. Aussi nous présentons les divers résultats de cette thèse en HOAS par souci de clarté, sachant que le développement formel, lui, justifie ces résultats en PHOAS pour éviter les paradoxes. Nous employons également des PTS où les variables sont nommées pour plus de lisibilité et de concision dans les obligations de preuves, alors que le développement formel part de PTS avec indices de de Bruijn. Nous partons ainsi d'un système dont la métathéorie est bien comprise et vérifiée dans Coq (Barras, 1996).

4.6.2 Complétude

Dans le système de types standard pour les PTS, toutes les dérivations de typage vérifient la bonne formation du contexte aux feuilles.

Définition 4.27 Nous notons $\hat{\cdot}$ la fonction qui à un nom x associe son indice dans un certain contexte de typage, qui est un argument implicite à la fonction donné par le contexte discursif. Par exemple, $\hat{x}_1 = 0$ si l'on fixe le contexte à $\Gamma = x_1 : A_1, \ldots, x_n : A_n$.

Définition 4.28 (Traduction des λ^{pts}-termes) Nous définissons une traduction $[\![\cdot]\!]$, des λ^{pts}-termes M clos par un environnement ρ associant un terme à toutes les variables libres dans M, vers les $\lambda^{\mathsf{pts}}_{bn}$-termes ainsi :

$$
\begin{aligned}
[\![s]\!]\ \rho &= s \\
[\![x]\!]\ \rho &= \rho(x) && \text{si } x \in \mathsf{dom}(\rho) \\
[\![x]\!]\ \rho &= \hat{x} && \text{sinon} \\
[\![\lambda x : A.\ M]\!]\ \rho &= \lambda x : A.\ [\![M]\!]\ \rho[x \mapsto x] \\
[\![\Pi x : A.\ B]\!]\ \rho &= \Pi x : A.\ [\![B]\!]\ \rho[x \mapsto x] \\
[\![M\ N]\!]\ \rho &= ([\![M]\!]\ \rho)\ ([\![N]\!]\ \rho)
\end{aligned}
$$

Cette traduction est une fonction partielle. Elle n'est pas définie pour les termes non clos par ρ.

Lemme 4.29 (Commutation de la substitution pour l'encodage) *Pour toute variable x, environnement ρ et termes M,N,*

$$\ulcorner [\![\{N/x\}M]\!]\ \rho \urcorner = (\overline{\lambda} y.\ \ulcorner [\![M]\!]\ \rho[x \mapsto y] \urcorner)\ \ulcorner [\![N]\!]\ \rho \urcorner.$$

Preuve. Par induction sur M. Dans le cas où $M = \lambda\ A\ (\overline{\lambda} z.\ M')$, on cherche à prouver

$$\lambda \ulcorner[\![\{N/x\}A]\!]\ \rho\urcorner\ (\overline{\lambda}z.\ \ulcorner[\![\{N/x\}M']\!]\ \rho[z \mapsto z]\urcorner) =$$
$$\lambda \ulcorner[\![A]\!]\ \rho[x \mapsto \ulcorner[\![N]\!]\ \rho\urcorner]\urcorner\ (\overline{\lambda}z.\ \ulcorner[\![M']\!]\ \rho[z \mapsto z][x \mapsto \ulcorner[\![N]\!]\ \rho\urcorner]\urcorner).$$

Il est nécessaire ici d'utiliser l'axiome d'extensionnalité pour passer sous l'abstraction $(\overline{\lambda}z.\ \ldots)$ du métalangage et réécrire le corps de l'abstraction avec l'hypothèse d'induction pour M'. Le cas où M est un produit dépendant est similaire. □

Lemme 4.30 *Pour tous λ^{pts}-termes M ,M' tel que $M \rightrightarrows_\beta M'$,*

$$\ulcorner[\![M]\!]\ \Gamma^*\urcorner \rightrightarrows_\beta^n \ulcorner[\![M']\!]\ \Gamma^*\urcorner.$$

Preuve. Nous procédons par induction mutuelle sur la dérivation de $M \rightrightarrows_\beta M'$ et Γ. Il nous faut généraliser l'énoncé pour renforcer l'hypothèse d'induction :

$$M \rightrightarrows_\beta M' \Rightarrow \ulcorner[\![M]\!]\ (\rho +\!\!+ \Gamma^*)\urcorner \rightrightarrows_\beta^n \ulcorner[\![M']\!]\ (\rho +\!\!+ \Gamma^*)\urcorner.$$

où ρ est une liste de variables libres $m_1, \ldots, m_n$ toutes plus petites que n.
Dans le cas de la β-réduction, M est de la forme $\lambda x :\ A.\ M_1\ M_2$, $M' = \{M_2'/x\}M_1'$, $M_1 \rightrightarrows_\beta M_1'$, et $M_2 \rightrightarrows_\beta M_2'$. Soit $A_h = \ulcorner[\![A]\!]\ (\rho +\!\!+ \Gamma^*)\urcorner$ et $M_h' = \ulcorner[\![\{M_2'/x\}M_1']\!]\ (\rho +\!\!+ \Gamma^*)\urcorner$. Nous cherchons à prouver

$$(\lambda\ A_h\ (\overline{\lambda}x.\ \ulcorner[\![M_1]\!]\ (\rho[x \mapsto x] +\!\!+ \Gamma^*)\urcorner)) \cdot \ulcorner[\![M_2]\!]\ (\rho +\!\!+ \Gamma^*)\urcorner \rightrightarrows_\beta^n M_h'.$$

Par le lemme 4.29, $M_h' = \ulcorner[\![M_1']\!]\ (\rho[x \mapsto \ulcorner[\![M_2']\!]\ (\rho +\!\!+ \Gamma^*)\urcorner] +\!\!+ \Gamma^*)\urcorner$, que nous pouvons abstraire en $M_h' = (\overline{\lambda}x.\ \ulcorner[\![M_1']\!]\ (\rho[x \mapsto x] +\!\!+ \Gamma^*)\urcorner)\ \ulcorner[\![M_2']\!]\ (\rho +\!\!+ \Gamma^*)\urcorner$. Nous pouvons maintenant prouver le résultat par la dérivation suivante, en commençant par les hypothèses d'induction :

$$\frac{\ulcorner[\![M_1]\!]\ \rho'\urcorner \rightrightarrows_\beta^{n+1} \ulcorner[\![M_1']\!]\ \rho'\urcorner \qquad \ulcorner[\![M_2]\!]\ (\rho +\!\!+ \Gamma^*)\urcorner \rightrightarrows_\beta^n \ulcorner[\![M_2']\!]\ (\rho +\!\!+ \Gamma^*)\urcorner}{(\lambda\ A_h\ (\overline{\lambda}x.\ \ulcorner[\![M_1]\!]\ \rho'\urcorner)) \cdot \ulcorner[\![M_2]\!]\ (\rho +\!\!+ \Gamma^*)\urcorner \rightrightarrows_\beta^n M_h'}$$

où $\rho' = \rho[x \mapsto x] +\!\!+ \Gamma^*$.
Dans le cas où M est une variable x, Il nous faut prouver

$$(\rho +\!\!+ \Gamma^*)(x) \rightrightarrows_\beta (\rho +\!\!+ \Gamma^*)(x).$$

Soit $x \in \rho$ ou alors $x \in \Gamma^*$. Dans le premier cas, $\rho(x)$ est une variable libre m, qui se réduit en elle-même. Dans le deuxième cas, par inversion $\Gamma^*(x)$ est de la forme $[m : A]$, qui encore une fois se réduit en lui-même.
Les autres cas ne posent pas de difficultés particulières. □

Corollaire 4.31 *Pour tous λ^{pts}-termes M ,M' tel que $M \equiv_\beta M'$,*

$$\ulcorner [\![M]\!]\ \Gamma^* \urcorner \equiv^n_\beta \ulcorner [\![M']\!]\ \Gamma^* \urcorner.$$

Définition 4.32 Nous construisons un environnement ρ à partir d'un contexte de typage Γ par la fonction suivante :

$$[]^* = [] \qquad\qquad (\Gamma, x : A)^* = \Gamma^*[x \mapsto [\hat{x} : \ulcorner [\![M]\!]\ \Gamma^* \urcorner]]$$

Lemme 4.33 (Inversion de l'encodage des contextes) *Pour tout nom de variable x, terme M et contexte Γ tel que $x \mapsto M \in \Gamma^*$, il existe un type A tel que $M = [\hat{x} : \ulcorner [\![A]\!]\ \Gamma^* \urcorner]$.*

Preuve. Par induction sur Γ. □

Exemple 4.34 Soit $M = \Pi x : A.\ P\ x$ clos par un contexte $\Gamma = A : *, P : \Pi y : A.\ *$. Alors,

$$\begin{aligned}\ulcorner [\![M]\!]\ \Gamma^* \urcorner &= \ulcorner [\![M]\!]\ [A \mapsto [0 : *], P \mapsto [1 : \Pi\ [0 : *]\ (\overline{\lambda} y.\ *)]] \urcorner \\ &= \Pi\ [0 : *]\ (\overline{\lambda} x.\ [1 : \Pi\ [0 : *]\ (\overline{\lambda} y.\ *)] \cdot x)\end{aligned}$$

Lemme 4.35 *Si $\Gamma \vdash M : A$, alors $\ulcorner [\![M]\!]\ \Gamma^* \urcorner$ est bien défini.*

Preuve. Par induction sur la dérivation de $\Gamma \vdash M : A$. □

Théorème 4.36 (Complétude) *Pour tout contexte Γ, termes M ,A tel que $\Gamma \vdash M : A$ et Γ wf,*

$$\vdash_n \ulcorner [\![M]\!]\ \Gamma^* \urcorner : \ulcorner [\![A]\!]\ \Gamma^* \urcorner$$

où n est la taille de Γ et donc $\forall x.\ x \in \mathsf{dom}(\Gamma^*) \Rightarrow \hat{x} < n$.

Preuve. Soit H la dérivation de $\Gamma \vdash M : A$ et W une dérivation de Γ wf. Nous procédons par induction mutuelle sur H et W .

cas $(sort)$: immédiat.

cas (var) : par le lemme 4.33, $\Gamma^*(x)$ est de la forme $[\hat{x} : \ulcorner [\![A]\!]\ \Gamma^* \urcorner]$. Il nous faut donc une dérivation de $\exists s.\ \vdash_{\hat{x}} \ulcorner [\![A]\!]\ \Gamma^* \urcorner : s$. Par bonne formation du contexte une telle dérivation existe puisque $x : A \in \Gamma$ donc $\exists \Gamma'. \exists s.\ \Gamma' \vdash A : s$ et ainsi $\vdash_{n'} \ulcorner [\![A]\!]\ \Gamma'^* \urcorner : s$ où n' est la taille de Γ'. $n' < \hat{x}$ par bonne formation du contexte et donc $\exists s.\ \vdash_{\hat{x}} \ulcorner [\![A]\!]\ \Gamma^* \urcorner : s$. Nous pouvons donc former la dérivation suivante :

$$\frac{\vdash_{\hat{x}} \ulcorner [\![A]\!]\ \Gamma^* \urcorner : s}{\vdash_n [\hat{x} : \ulcorner [\![A]\!]\ \Gamma^* \urcorner]}$$

$$\frac{}{\Gamma \vdash s \simeq s}\,(psort) \qquad \frac{(M,N) \in \Gamma}{\Gamma \vdash M \simeq N}\,(pmem) \qquad \frac{}{\Gamma \vdash m \simeq m}\,(pfvar)$$

$$\frac{\Gamma \vdash A_1 \simeq A_2}{\Gamma \vdash [m : A_1] \simeq [m : A_2]}\,(pcast)$$

$$\frac{\Gamma \vdash A_1 \simeq A_2 \qquad \forall N_1.\forall N_2.\ \Gamma, (N_1,N_2) \vdash M_1\ N_1 \simeq M_2\ N_2}{\Gamma \vdash \lambda\ A_1\ M_1 \simeq \lambda\ A_2\ M_2}\,(pabs)$$

$$\frac{\Gamma \vdash A_1 \simeq A_2 \qquad \forall N_1.\forall N_2.\ \Gamma, (N_1,N_2) \vdash B_1\ N_1 \simeq B_2\ N_2}{\Gamma \vdash \Pi\ A_1\ B_1 \simeq \Pi\ A_2\ B_2}\,(pprod)$$

$$\frac{\Gamma \vdash M_1 \simeq M_2 \qquad \Gamma \vdash N_1 \simeq N_2}{\Gamma \vdash M_1 \cdot N_1 \simeq M_2 \cdot N_2}\,(papp)$$

Figure 4.7 Relation de paramétricité. Γ est une liste de couples de termes.

cas (abs) : par hypothèses d'induction.
cas $(prod)$: par hypothèses d'induction.
cas (app) : par les hypothèses d'induction et le lemme 4.29.
cas $(conv)$: conséquence du lemme 4.29.

□

4.6.3 Correction

La preuve de correction procède selon les mêmes grandes lignes que la preuve de correction pour le système de types hors contexte pour le λ^{st}-calcul. Comme dans la section 4.2.2, les noms de variables sont étiquetés par un type pour éviter les hypothèses incohérentes de typage.

Nous ne pouvons relier les PTS standards sur des termes du premier ordre aux PTS hors contexte sur des termes en HOAS sans nous arrêter d'abord sur l'adéquation des termes en HOAS par rapport aux termes usuels au premier ordre. Nous ne pouvons espérer prouver la correction des PTS hors contexte sans nous assurer d'abord que les représentations de termes que nous avons sous la main correspondent vraiment à des termes — qu'elles ne sont pas des valeurs exotiques qui ne trouvent pas leur pendant dans l'ensemble des termes des PTS standards. Intuitivement, il s'agit d'imposer une condition nécessaire et suffisante sur l'espace des fonctions servant à représenter le corps des abstractions et des produits dépendants.

Définition 4.37 Un $\lambda^{\mathsf{pts}}_{bh}$-terme M est dit *paramétrique* si $[] \vdash M \simeq M$. Les dérivations de ce jugement sont données inductivement par les règles de la figure 4.7.

L'idée derrière la condition de paramétricité est que les fonctions représentant le corps des abstractions et des produits dépendants ne doivent pas donner de résultats qui ne sont pas reliés à partir du moment où elles sont appliquées à des arguments qui sont reliés.

La relation que nous donnons dans la figure 4.7 définit la notion de paramétricité sur les termes en HOAS. Nous supposerons dans le reste de cette section que tous les termes en HOAS sont paramétriques. Dans la formalisation en Coq de cette section, nous utilisons une relation similaire, adaptée aux termes en PHOAS, que nous empruntons à Chlipala (2008). Ce dernier postule que tous les termes en PHOAS du bon type sont paramétriques et énonce un axiome à cet effet. Nous observons dans notre développement que tous les termes en PHOAS images d'une certaine traduction de termes avec indices de de Bruijn sont paramétriques, évitant ainsi le recours à un axiome. De même, nous montrons ici que tout terme en HOAS image de la traduction définie dans la section 4.6.2 est paramétrique.

Définition 4.38 Soit deux environnements ρ,σ tel que ρ est plus petit que σ. Nous définissons la convolution $\rho * \sigma$ de ρ et σ par induction sur ρ :

- si $\rho = []$ alors $\rho * \sigma = []$;
- si $\rho = \rho'[x \mapsto M\,]$ alors $\rho * \sigma = \rho' * \sigma, (\rho(x), \sigma(x))$.

Théorème 4.39 *Soit un terme M clos par deux environnements ρ et σ. Alors,*

$$\rho * \sigma \vdash \ulcorner [\![M]\!]\ \rho \urcorner \simeq \ulcorner [\![M]\!]\ \sigma \urcorner.$$

Preuve. Par induction sur M, en remarquant que pour tous x,M_1,M_2,

$$\rho * \sigma, (M_1,M_2) = (\rho[x \mapsto M_1] * \sigma[x \mapsto M_2]).$$

□

Corollaire 4.40 *La représentation $\ulcorner [\![M]\!]\ [] \urcorner$ de tout terme clos M est paramétrique.*

Il sera parfois nécessaire dans cette section de procéder par induction sur les termes encodés en HOAS, ou plus souvent sur la dérivation des jugements de typage des termes encodés en HOAS. L'induction sur ces termes, comme l'induction sur ces dérivations, pose problème dans les théories des types usuelles que nous prenons comme cadre de raisonnement ici. Comme le corps des abstractions et des produits dépendants sont représentés comme des fonctions, les hypothèses d'induction concernant le corps des abstractions et des produits dépendants sera de la forme $M\ N$ où N est un terme quelconque pour remplacer les occurrences de la variable devenue libre dû au passage sous le lieur. Mais étant donné un prédicat P, il nous faudra souvent établir que

$$P\ (M\ N)\ (M\ N')$$

pour une certain autre terme N'. Dans une représentation du premier ordre, le but ci-dessus aura la forme

$$P\ (\{N/x\}M)\ (\{N'/x\}M),$$

que l'on démontrera typiquement par induction sur M. Mais dans le but précédent M est une fonction. Dans la logique de Coq, le CCI, il n'est pas possible de faire une preuve par induction sur une fonction puisque le type d'une fonction n'est pas un type inductif[32].

Mais une induction sur le jugement de paramétricité du terme suffit. Nous l'illustrerons avec une preuve que la substitution sur les $\lambda^{\mathsf{pts}}_{bh}$-termes correspond à la substitution sur les λ^{pts}-termes (lemme 4.45).

Définition 4.41 Nous définissons une fonction de traduction $(\downarrow_n)$ des $\lambda^{\mathsf{pts}}_{bh}$-termes vers les $\lambda^{\mathsf{pts}}_{bn}$-termes paramétrée par un entier n :

$$\downarrow_n s = s \qquad \downarrow_n \lambda\ A\ M = \lambda n^{\downarrow_n A} : (\downarrow_n A).\ \downarrow_{n+1} (M\ [n : A])$$

$$\downarrow_n [m : A] = m^{\downarrow_n A} \qquad \downarrow_n \Pi\ A\ B = \Pi n^{\downarrow_n A} : (\downarrow_n A).\ \downarrow_{n+1} (B\ [n : A])$$

$$\downarrow_n M \cdot N = (\downarrow_n M)\ (\downarrow_n N)$$

Cette fonction de traduction est d'une similarité frappante avec la fonction de réification du chapitre 3. La différence majeure réside dans la propagation du type des variables libres avec celles-ci. Nous aurions pu choisir de réifier le corps d'une abstraction en appliquant la fonction du métalangage tenant lieu de corps à une simple variable libre plutôt qu'une annotation, mais la définition ci-dessus nous sera d'une grande utilité plus tard.

Il ne reste plus qu'à collectionner les annotations de typage pour former un contexte.

Définition 4.42 (Fusion de deux contextes) Soit deux contextes Γ,Δ. La fusion de ces deux contextes, noté $\Gamma \ltimes \Delta$, est définie par

$$\Gamma \ltimes \Delta = \Gamma, (\Delta \setminus \Gamma).$$

Définition 4.43 (Collection des annotations) La collection des annotations dans un $\lambda^{\mathsf{pts}}_{bh}$-terme est donné comme suit :

[32] D'autres systèmes tels que Twelf font le choix d'une logique très pauvre où tous les termes sont systématiquement en forme β-normale et η-longue. En contrepartie, ce choix facilite l'implémentation de mécanismes semi-automatisés de méta-raisonnement par induction sur la forme des termes.

$$\#(s)_n = [] \qquad \#(\lambda\ A\ M)_n = \#(M\ [n : A])_{n+1} \setminus n : \mathord{\downarrow}_n A$$
$$\#([m : A])_n = \#(A)_m, m : \mathord{\downarrow}_n A \qquad \#(\Pi\ A\ B)_n = \#(B\ [n : A])_{n+1} \setminus n : \mathord{\downarrow}_n A$$
$$\#(M\ \cdot\ N)_n = \#(M)_n \ltimes \#(N)_n$$

La collection d'une annotation fait ce que la collection dans une abstraction ou un produit défait : une annotation introduit une nouvelle hypothèse dans un contexte qui est filtré plus tard au niveau d'une abstraction ou d'un produit. Contrairement au λ-calcul simplement typé, il est nécessaire ici de combiner des contextes dont les hypothèses sont disjointes, pour éviter de compromettre la bonne formation du contexte obtenu.

Définition 4.44 (Entier majorant) Un entier n est dit *majorant* pour une liste de $\lambda_{bh}^{\mathsf{pts}}$-termes $M_1, \ldots, M_n$ si

$$\forall m.\ m \in \bigcup_i \mathcal{FV}(M_i) \Rightarrow m < n.$$

Lemme 4.45 *Pour tout termes M,N,A et contexte Γ tel que*

1. *la taille n de Γ est un entier majorant pour M,N et A ;*
2. $\forall M_1.\forall M_2.\ (M_1,M_2) \in \Gamma \Rightarrow \mathord{\downarrow}_{n+1} M_1 = \{\mathord{\downarrow}_n N/n\}(\mathord{\downarrow}_{n+1} M_2)$.

Alors nous avons

$$\mathord{\downarrow}_{n+1} (M\ N) = \{\mathord{\downarrow}_n N/n\}(\mathord{\downarrow}_{n+1} (M_2\ [n : A])).$$

Preuve. Par induction sur la dérivation de $\Gamma \vdash M\ N \simeq M\ [n : A]$. Nous détaillons ici les cas $(pmem)$, $(pcast)$ et $(pabs)$. Les autres cas sont similaires ou immédiats.

cas $(pmem)$: alors $M\ N = M_1$ et $M\ [n : A] = M_2$ tel que $(M_1,M_2) \in \Gamma$ et donc $\mathord{\downarrow}_{n+1} M_1 = \{\mathord{\downarrow}_n N/n\}(\mathord{\downarrow}_{n+1} M_2)$ par hypothèse.

cas $(pcast)$: alors $M_1 = [m : A_1]$ et $M_2 = [m : A_2]$. Comme $m < n$, nous obtenons immédiatement le résultat $m = \{\mathord{\downarrow}_{n+1} N/n\}m$.

cas $(pabs)$: alors $M\ N$ est de la forme $\lambda\ A_1\ M_1$ et $M\ [n : A]$ est de la forme $\lambda\ A_2\ M_2$. Il nous suffit donc de montrer que

$$\mathord{\downarrow}_{n+1} A_1 = \{\mathord{\downarrow}_n N/n\}(\mathord{\downarrow}_{n+1} A_1)$$

et que

$$\mathord{\downarrow}_{n+2} (M_1\ (n+1)) = \{\mathord{\downarrow}_n N/n+1\}\ \mathord{\downarrow}_{n+2} (M_2\ [(n+1) : A]).$$

Ces deux équations sont des instances des deux hypothèses d'induction.

□

Corollaire 4.46 *Soit deux $\lambda_{bh}^{\mathsf{pts}}$-termes M,M' et un entier majorant n. Si $M \equiv_\beta M'$ alors $\downarrow_n M \equiv_\beta \downarrow_n M'$.*

Lemme 4.47 *Pour tous termes M,A, si $\vdash_n M : A$ alors n est un entier majorant.*

Preuve. Par induction sur la dérivation de $\vdash_n M : A$. □

Lemme 4.48 *Pour tous $\lambda_{bh}^{\mathsf{pts}}$-termes M,A et deux entiers majorants n,m tels que $m < N$,*

$$\vdash_m M : A \Rightarrow \vdash_n M : A.$$

Preuve. Par induction sur la dérivation de $\vdash_m M : A$ et paramétricité des termes. □

Lemme 4.49 *Pour tout terme M et entier m,n tels que $m < n$,*

$$\#(A)_m \ \mathsf{wf} \Rightarrow \#(A)_n \ \mathsf{wf}.$$

Preuve. Par induction sur la dérivation de $\#(A)_m$ wf. □

Lemme 4.50 *Pour tous termes M,A et entiers m,n tels que $m < n$, si $\vdash_m M : A$ alors $m \notin \#(M)_n$ et $m \notin \#(A)_n$.*

Preuve. Par induction sur la dérivation de $\vdash_m M : A$. □

Théorème 4.51 (Correction) *Soit deux $\lambda_{bh}^{\mathsf{pts}}$-termes M,A et un entier n majorant.*

$$\textit{Si } \vdash_n M : A \textit{ alors } \#(A)_n \ltimes \#(M)_n \vdash \downarrow_n M : \downarrow_n A.$$

Preuve. Par induction sur la dérivation de $\vdash_n M : A$.

Cas ($cast_{hn}$) : alors M est de la forme $[m : A]$ et $m < n$. Le contexte $\#(A)_m$ est bien formé puisque nous avons comme hypothèse d'induction $\#(A)_m \vdash \downarrow_m A : s$. $\#(A)_n$ est donc lui aussi bien formé puisque $m < n$, par le lemme 4.49. De plus, $\vdash_m A : s$ donc $m \notin \#(A)_n$ par le lemme 4.50 et ainsi le contexte $\#(A)_n, m : \downarrow_n A$ est bien formé. Nous avons par conséquent que

$$\#(A)_n, m : \downarrow_n A \vdash m : \downarrow_n A.$$

Cas ($sort_{hn}$) : immédiat.

Cas (abs_{hn}) : Nous avons comme hypothèses d'induction :

$$\#(B\ [n:A])_{n+1} \ltimes \#(M\ [n:A])_{n+1} \vdash \downarrow_{n+1} (M\ [n:A]) : \downarrow_{n+1} (B\ [n:A]) \quad (4.5)$$

$$\#(s)_n \ltimes \#(\Pi\ A\ B)_n \vdash \downarrow_n \Pi\ A\ B : \downarrow_n s \quad (4.6)$$

Si $n\ {}^{\downarrow_n A} : \downarrow_n A \in \#(B\ [n:A])_{n+1} \ltimes \#(M\ [n:A])_{n+1}$ alors aucune autre hypothèse de ce contexte ne dépend de $n\ {}^{\downarrow_n A}$ puisque toutes les hypothèses du contexte ne peuvent dépendre que d'autres hypothèses dont le numéro est plus petit et toutes les hypothèses ont un numéro inférieur ou égal à n. Nous pouvons donc, par permutation, déplacer cette hypothèse à la fin du contexte :

$$\Gamma = \#(B\ [n:A])_{n+1} \setminus n\ {}^{\downarrow_n A} : \downarrow_n A \ltimes \#(M\ [n:A])_{n+1} \setminus n\ {}^{\downarrow_n A} : \downarrow_n A, n\ {}^{\downarrow_n A} : \downarrow_n A.$$

Par ailleurs, le jugement (4.6) peut s'écrire

$$\Gamma \vdash \Pi n\ {}^{\downarrow_n A} : \downarrow_n A.\ \ \downarrow_{n+1} (B\ [n:A]) : \downarrow_n s$$

par affaiblissement. Ainsi, par la règle (abs), nous avons une dérivation du jugement

$$\Gamma \vdash \lambda n\ {}^{\downarrow_n A} : \downarrow_n A.\ \ \downarrow_{n+1} (M\ [n:A]) : \Pi n\ {}^{\downarrow_n A} : \downarrow_n A.\ \ \downarrow_{n+1} (B\ [n:A]).$$

En remarquant que $\Gamma = \#(\Pi\ A\ B)_n \ltimes \#(\lambda\ A\ M)_n$, nous obtenons le résultat.

Cas ($prod_{hn}$) : similaire au cas (abs_{hn}).

Cas (app_{hn}) : par le lemme 4.45.

Cas ($conv_{hn}$) : par le corollaire 4.46.

□

4.7 Vers un vérificateur de types

Nous nous tournons vers l'implémentation d'un vérificateur de types pour les termes des PTS encodés en HOAS qui s'affranchit de tout contexte explicite. Nous avons maintenant un système formel hors contexte avec lequel nous pouvons justifier les jugements de typage. Mais la distance entre les dérivations de typage de ce système et un algorithme de décision pour un jugement de la forme $\vdash_n M : A$ est encore grande : comme pour les PTS standards, le système hors contexte n'est pas dirigé par la syntaxe du terme M dont il faut vérifier le type.

Nous avons vu dans le chapitre 2 qu'un algorithme de décision pour tous les PTS dans toute leur généralité n'existe pas. Par simplicité, nous ne considérons dans le reste de ce chapitre que les PTS fonctionnels, qui admettent aisément une présentation dirigée par la syntaxe. Une généralisation à d'autres classes de PTS, tels que les PTS semi-pleins ou les PTS cumulatifs, ne devrait pas poser de problèmes particuliers au delà de ceux déjà identifiés par van Benthem Jutting et al. (1993).

4.7.1 Inférence de types

Considérons un système de types comme celui de la section 4.6 sans règle de conversion. Un tel système de types n'admet pas de calcul au niveau des types et est déjà dirigé par la syntaxe si les axiomes $\mathcal{A}$ et les règles $\mathcal{R}$ du PTS sont fonctionnels. Mais étant donné un terme M avec un entier majorant n, on ne peut calculer efficacement son type. Si M est une abstraction de la forme $\lambda\ A\ M'$, il nous faut entre autre construire une fonction B pour former le produit dépendant $\Pi\ A\ B$. Inférer le type du corps $M'\ [n : A]$ nous donne un terme C, qu'il faut pouvoir écrire sous la forme d'une fonction B appliquée à une annotation $[n : A]$. Nous pouvons obtenir cette fonction par abstraction de toutes les occurrence de $[n : A]$ dans C, c'est à dire en faisant l'inverse de la substitution. Plus formellement, la clause de la fonction d'inférence of traitant des abstractions serait de la forme :

$$\mathsf{of}\ n\ (\lambda\ A\ M) = \Pi\ A\ (\overline{\lambda}x.\ \mathsf{abstract}\ [n : A]\ x \\ (\mathsf{of}\ (n+1)\ (M\ [n : A]))) \qquad \text{si } \mathsf{of}\ n\ A = s$$

En pratique, il est plus simple et plus efficace de procéder ainsi :

$$\mathsf{of}\ n\ (\lambda\ A\ M) = \Pi\ A\ (\overline{\lambda}x.\ \mathsf{of}\ n\ (M\ x)) \qquad \text{si } \mathsf{of}\ n\ A = s$$

Nous pouvons voir cette dernière formulation comme une déforestation (Wadler, 1990), c'est à dire placer un raccourci à la place d'un détour[33]. Le détour est ici une instanciation de M à $[n : A]$ suivie d'une abstraction sur les occurrences de $[n : A]$. Nous remplaçons une clause où l'on trouve une « introduction » de $[n : A]$ suivie de son « élimination » par une version plus directe. La dérivation de typage correspondant à cette clause est la suivante :

$$(abs'_{hn})\ \frac{\forall x.\ \vdash_n x : A \Rightarrow\ \vdash_n M\ x : B\ x \qquad \vdash_n \Pi\ A\ B : s}{\vdash_n \lambda\ A\ M : \Pi\ A\ B} \tag{4.7}$$

M est de type B pour toute instanciation x de type A et non plus seulement $[n : A]$[34]. On peut aussi voir la règle (abs_{hn}) comme une internalisation d'un lemme de substitution pour les PTS hors contexte en HOAS. L'équivalence entre les deux systèmes découle en effet directement de ce lemme.

Définition 4.52 Étant donné deux termes M,A, nous notons les jugements du système de type de la section 4.6 dont la règle (abs_{hn}) a été remplacé par (abs'_{hn}) par $\vdash'_n M : A$.

[33] Ou en termes de la théorie des preuves : l'élimination d'une « coupure ».

[34] La fonction of peut se passer de la vérification que x doit être de type A si l'on montre que of n'applique jamais les fonctions représentant les corps des abstractions et des produits dépendants à des termes d'un type autre que leur domaine.

Lemme 4.53 (Substitution pour les PTS hors contexte) *Soit deux termes M,A et un entier n. Si $\vdash_n M : A$ alors $\vdash'_n M : A$.*

Preuve. Nous pouvons obtenir ce résultat de deux manières différentes : soit directement, soit en exploitant la correspondance entre les PTS du chapitre 2 et les PTS hors contexte et le lemme de substitution déjà établi pour le système standard. □

Corollaire 4.54 *Soit M,A,B des termes et n un entier.*

$$\textit{Si } \vdash_{n+1} M\ [n : B] : A \textit{ alors } \forall N\,.\ \vdash_n N : B \Rightarrow \vdash_n M\ N : B\ N\,.$$

4.7.2 Le typage est statique, le calcul est dynamique

Le système de type de la section précédente réduit plus encore la distance entre le système formel et un algorithme décidant de la bonne typabilité d'un terme. Mais tous les systèmes jusqu'à présent restent peu diserts sur l'interaction entre l'implémentation du test de conversion et le reste du typage. En effet, un terme d'un PTS est traité comme un morceau de syntaxe dont on vérifie le type un instant, puis comme un objet calculatoire quelques instants suivants, lors du test de conversion. Comme nous l'avons vu dans le chapitre 3, faire un test de conversion demande typiquement de calculer la forme normale de termes. Les termes auront avantage à être traduits dans un autre format (du code compilé par exemple) pour trouver cette forme normale, plutôt que d'interpréter directement ces morceaux de syntaxe.

En pratique, le typage jongle donc entre deux représentations des termes. Nous dirons que la représentation des termes utile au typage est la représentation « statique », alors que la représentation employée lors du calcul est dite « dynamique ». En plus d'être des termes compilés, les termes dynamiques sont typiquement des termes à la Curry, puisque les types sont inutiles lors du calcul. Barthe et Sørensen (2000) montrent que comparer des termes à la Curry (sans domaine) plutôt que des termes à la Church des PTS fonctionnels normalisants ne change pas la théorie de ces systèmes. Barras et Grégoire (2005) généralisent ce résultat aux PTS cumulatifs et à l'effacement des paramètres des types inductifs.

La figure 4.8 montre un système de types hors contexte dirigé par la syntaxe sur des termes à deux niveaux, statiques ou dynamiques. Nous supposons que les termes dynamiques sont représentés comme dans le chapitre 3. Dans cette section, nous désignerons la classe des termes étiquetés du chapitre 3 par $\overline{\mathsf{Term}}$ et utiliserons les métavariables $\overline{M}, \overline{N}, \overline{A}, \overline{B}$ pour désigner de tels termes. Nous écrirons $\overline{[\![\cdot]\!]}$ l'interprétation des termes donnée dans le chapitre 3.

Intuitivement, les termes statiques sont une entrée du système de types et les sorties sont des types dynamiques, puisque dans un système dirigé par la syntaxe les types sont réduits à

la volée. Si tel est le cas, alors dans la règle (abs) par exemple, le domaine d'une abstraction apparaît sous forme statique dans le terme à typer, mais aussi sous forme dynamique dans le type produit. De même, l'argument d'une application est un terme statique, que l'on doit substitutuer dans un terme dynamique. Il paraît donc nécessaire d'écrire des coercions entre termes statiques et dynamiques, et inversement. Or les deux formes ne sont pas en bijection, puisque la forme dynamique d'un terme oublie le domaine des abstractions. On ne peut donc reconstruire un terme statique à partir d'un terme dynamique. Écrire des coercions est une solution d'autant plus inélégante qu'elle ne paraît seulement possible à l'aide d'une structure d'environnement — il serait dommage de laisser ce genre de structure s'approcher en si bon chemin.

Un artifice que l'on serait tenté d'employer serait d'adapter la fonction de traduction afin d'embarquer à la fois des formes statiques et des formes dynamiques dans l'encodage d'un terme, notamment pour le domaine des abstractions. Cette approche présente un gros inconvénient : les sous-termes étant dupliqués, la taille de l'encodage des termes peut être exponentiellement plus grande que le terme de départ. Prenons par exemple un ensemble de variables $a_1, \ldots, a_n$, où le type de chaque a_i est de la forme $\Pi x_i : A_i.\ B_i$. Alors la traduction du terme

$$a_1\ (a_2\ \ldots (a_{n-1}\ a_n)\ldots)$$

contiendrait $O(2^n)$ noeuds, puisque chaque A_i serait dupliqué.

Notons que tous les systèmes à base de types dépendants pâtissent du même problème du moment que ceux-ci compilent les termes à la volée pour les besoins de conversion de types[35]. Imaginons qu'il faille utiliser la règle de conversion pour vérifier que chaque A_i est convertible à chaque B_{i+1} durant la vérification du terme ci-dessus. À chaque noeud d'application, il faudra compiler chaque paire A_i,B_{i+1} vers du code natif pour décider la convertibilité efficacement. Si l'on suppose plus encore que chaque x_i apparaît libre dans chaque B_i, alors chaque a_i sera compilé $n - i$ fois, à cause de la dépendance de B_i sur x_i. La quantité totale de code produit sera donc proportionelle au carré de la taille du terme originel.

Ce problème est un problème de partage des sous-termes. Il faut pouvoir donner un nom à chaque sous-terme afin de pouvoir le partager entre différentes instances du test de conversion. La solution que nous proposons ici est de traduire le terme donné en entrée vers une forme « séquentialisée », plus restreinte, où tous les termes intermédiaires sont nommés.

[35] C'est notamment le cas lorsque l'on utilise la VM de Coq(Grégoire et Leroy, 2002)

$$(cast_{hnsyn})\ \frac{\vdash_m \hat{A} : s}{\vdash_n [m : A] : \langle \hat{A}, \check{A} \rangle}\ m < n \qquad (sort_{hnsyn})\ \frac{}{\vdash_n s_1 : \langle s_2,s_2 \rangle}\ \text{si } \langle s_1,s_2 \rangle \in \mathcal{A}$$

$$(abs_{hnsyn})\ \frac{\forall x.\ \vdash_n x : \langle A_0, \check{A} \rangle \Rightarrow\ \vdash_n M\ \hat{x} : \langle B\ \hat{x}, \overline{B}\ \check{x} \rangle \qquad \vdash_n \Pi\ \hat{A}\ B : \langle s,s \rangle}{\vdash_n \lambda\ A\ M : \langle \Pi\ \hat{A}\ B, \Pi\ \check{A}\ \overline{B} \rangle}$$

$$(prod_{hnsyn})\ \frac{\vdash_n \hat{A} : \langle s_1,s_1 \rangle \qquad \vdash_{n+1} M\ \langle [n : A], n \rangle : \langle s_2,s_2 \rangle}{\vdash_n \Pi\ A\ M : \langle s_3,s_3 \rangle}\ \text{si } \langle s_1,s_2,s_3 \rangle \in \mathcal{R}$$

$$(app_{hnsyn})\ \frac{\vdash_n \hat{M} : \langle \Pi\ A\ B, \Pi\ \overline{A}\ \overline{B} \rangle \qquad \vdash_n \hat{N} : \langle A_0, \overline{A} \rangle}{\vdash_n M \cdot N : \langle B\ N, \overline{B}\ \check{N} \rangle}$$

$$(let_{hnsyn})\ \frac{\vdash_n \hat{N} : A \qquad \vdash_{n+1} M\ \langle [n : A], n \rangle : B}{\vdash_n \textbf{let}\ N\ \textbf{in}\ M : B}$$

Figure 4.8 Système de type hors contexte dirigé par la syntaxe pour un PTS donné par un tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$, l'ensemble des sortes, l'ensemble des axiomes et l'ensemble des règles.

4.7.3 Typage de formes monadiques

Définition 4.55 ($\lambda_{bh}^{\mathsf{pts}}$-termes en forme monadique) Un $\lambda_{bh}^{\mathsf{pts}}$-terme M est en forme monadique s'il est de la forme suivante :

$$\begin{array}{rcccl} \mathsf{Atom_{bh}} & \ni & a, b & ::= & x \mid s \mid [m : a] \\ \mathsf{Fun} & \ni & f, g & ::= & \overline{\lambda} x.\ M \\ \mathsf{Value_{bh}} & \ni & v, w & ::= & a \mid \mathsf{Lam}\ a\ f \mid \mathsf{Pi}\ a\ f \\ \mathsf{Term_{bh}^{M}} & \ni & M, N & ::= & v \mid \mathsf{App}\ a\ b \mid \mathsf{Let}\ N\ f \end{array}$$

Comme précédemment, nous réutiliserons parfois les métavariables M, N, A ou B pour désigner des éléments syntaxiques de la classe Fun des fonctions du métalangage sur les termes. Nous réutilisons également les mêmes notations telles que $\lambda\ a\ f$ pour $\mathsf{Lam}\ a\ f$ et introduisons la nouvelle notation $\textbf{let}\ x \Leftarrow N\ \textbf{in}\ \overline{\lambda} x.\ M$ pour $\mathsf{Let}\ N\ (\overline{\lambda} x.\ M)$.

Définition 4.56 ($\lambda_{bh}^{\mathsf{pts}}$-term apparié) Un $\lambda_{bh}^{\mathsf{pts}}$hybride est un terme pouvant contenir une construction d'*appariement*[36] $\langle M, \overline{M} \rangle$ associant un terme M avec sa sémantique, c'est à dire une forme statique avec une forme dynamique.

[36] Comme pour l'auto-réduction de Mogensen citée dans le chapitre 3, il s'agit d'une construction similaire à la construction de « glueing » (Lafont, 1988 appendice A).

$$\begin{array}{rcccl} \mathsf{Term_{bh}} & \ni & M, N, A, B & ::= & s \mid x \mid \mathsf{Lam}\ A\ f \mid \mathsf{Pi}\ A\ f \\ & & & & \mid \mathsf{App}\ M\ N \mid [n : A] \mid \langle M, \overline{M} \rangle \end{array}$$

Nous notons par $\hat{M}$ la première projection de $\langle M, \overline{M} \rangle$ et $\check{M}$ sa deuxième projection.

Définition 4.57 ($\lambda_{bh}^{\mathsf{pts}}$-terme apparié en forme monadique) La forme monadique n'est pas stable par réduction. Aussi est-il pratique de donner une forme plus souple, stable par réduction et qui permet des constructions d'appariement.

$$\begin{array}{rcccl} \mathsf{Value_{bh}} & \ni & v, w & ::= & x \mid s \mid [m : \langle v, \overline{v} \rangle] \mid \mathsf{Lam}\ \langle v, \overline{v} \rangle\ f \mid \mathsf{Pi}\ \langle v, \overline{v} \rangle\ f \mid \langle v, \overline{v} \rangle \\ \mathsf{Term_{bh}^{M}} & \ni & M, N & ::= & v \mid \mathsf{App}\ \langle\ v, \overline{v} \rangle \langle w, \overline{w} \rangle \mid \mathsf{Let}\ \langle\ v, \overline{v} \rangle f \end{array}$$

Définition 4.58 (Traduction des λ^{pts}-termes en $\lambda_{bh}^{\mathsf{pts}}$-termes appariés en forme monadique) Cette traduction est définie comme suit :

$$\begin{aligned} [\![x]\!]_\rho &= \rho(x) \qquad \text{si } x \in \mathsf{dom}(\rho). \\ [\![s]\!]_\rho &= \langle s, s \rangle \\ [\![\lambda x : A.\ M]\!]_\rho &= \mathsf{Let}\ [\![A]\!]_\rho\ (\overline{\lambda} y.\ \langle \mathsf{Lam}\ \hat{y}\ (\overline{\lambda} x.\ [\![M]\!]_{\rho[x \mapsto x]}), \mathsf{Lam}\ (\overline{\lambda} x.\ \overline{[\![M]\!]}) \rangle) \\ [\![\Pi x : A.\ B]\!]_\rho &= \mathsf{Let}\ [\![A]\!]_\rho\ (\overline{\lambda} y.\ \langle \mathsf{Pi}\ \hat{y}\ (\overline{\lambda} x.\ [\![B]\!]_{\rho[x \mapsto x]}), \mathsf{Pi}\ \check{y}\ (\overline{\lambda} x.\ \overline{[\![B]\!]}) \rangle) \\ [\![M\ N]\!]_\rho &= \mathsf{Let}\ [\![N]\!]_\rho\ (\overline{\lambda} y.\ \mathsf{Let}\ [\![M]\!]_\rho\ (\overline{\lambda} z.\ \langle \mathsf{App}\ \hat{y}\ \hat{z}, \mathsf{app}\ \check{y}\ \check{z} \rangle)) \end{aligned}$$

où y,z sont choisis frais. Elle est la combinaison de la traduction de la section 4.6 (donnant des termes statiques) et de la traduction de la section 3.3 (donnant des termes dynamiques), fusionnées avec une transformation en forme monadique comme dans la section 1.6.1.

Exemple 4.59 Considérons le λ^{pts}-terme suivant

$$(\lambda v : (\lambda x : \mathsf{nat}.\ \mathsf{vec}\ x)\ \mathsf{O}.\ v)\ \mathsf{nil}$$

dans le contexte

$$\mathsf{nat} : \mathbf{Type}, \mathsf{O} : \mathsf{nat}, \mathsf{vec} : \mathsf{nat} \to \mathbf{Type}, \mathsf{nil} : \mathsf{vec}\ \mathsf{O}$$

Après application des réductions commutatives pour obtenir une forme A-normale et simplification, la traduction du λ^{pts}-terme ci-dessus donne le $\lambda_{bh}^{\mathsf{pts}}$-terme apparié suivant :

$$\begin{array}{l}
\mathsf{Let}\ \langle \mathsf{Lam}\ \mathsf{nat}\ (\overline{\lambda}x.\ \mathsf{App}\ \mathsf{vec}\ x), \mathsf{Lam}\ (\overline{\lambda}x.\ \mathsf{app}\ \mathsf{vec}\ x)\rangle \\
(\overline{\lambda}x_1.\ \mathsf{Let}\ \langle \mathsf{App}\ \hat{x}_1\ \mathsf{O}, \mathsf{app}\ \check{x}_1\ \mathsf{O}\rangle \\
(\overline{\lambda}x_2.\ \mathsf{Let}\ \langle \mathsf{Lam}\ \hat{x}_2\ (\overline{\lambda}v.\ v), \mathsf{Lam}\ (\overline{\lambda}v.\ v)\rangle \\
(\overline{\lambda}x_3.\ \langle \mathsf{App}\ \hat{x}_3\ \mathsf{nil}, \mathsf{app}\ \check{x}_3\ \mathsf{nil}\rangle))))
\end{array}$$

que l'on peut aussi écrire, d'après la notation introduite précédemment,

$$\begin{array}{l}
\mathbf{let}\ x_1 \Leftarrow \langle \lambda\ \mathsf{nat}\ (\overline{\lambda}x.\ \mathsf{vec}\cdot x), \mathsf{Lam}\ (\overline{\lambda}x.\ \mathsf{app}\ \mathsf{vec}\ x)\rangle\ \mathbf{in} \\
\mathbf{let}\ x_2 \Leftarrow \langle \hat{x}_1 \cdot \mathsf{O}, \mathsf{app}\ \check{x}_1\ \mathsf{O}\rangle\ \mathbf{in} \\
\mathbf{let}\ x_3 \Leftarrow \langle \lambda\ \hat{x}_2\ (\overline{\lambda}v.\ v), \mathsf{Lam}\ (\overline{\lambda}v.\ v)\rangle\ \mathbf{in} \\
\langle \hat{x}_3 \cdot \mathsf{nil}, \mathsf{app}\ \check{x}_3\ \mathsf{nil}\rangle
\end{array}$$

Les termes et les types sur lesquelles portent les règles de typage de la figure 4.8 sont des $\lambda^{\mathsf{pts}}_{bh}$appariés en forme monadique. Dans cette forme, le domaine et le codomaine d'une abstraction sont toujours des paires, que l'on peut projeter. De même, les produits dépendants sont toujours des produits de paires et les membres gauche et droit d'une application sont aussi des paires.

La relation de typage relie toujours un terme statique avec un type statique apparié avec sa version dynamique. Nous avons ainsi toujours une version dymanique des types sous la main, en forme normale, que nous pouvons utiliser pour comparer les types plutôt que d'avoir une règle de conversion.

Lemme 4.60 *Soit Γ un contexte, M ,A,B des λ^{pts}-termes d'un PTS fortement normalisant et n la taille de Γ.*

$$\textit{(conv)}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}$$

si et seulement si

$$\vdash_n \ulcorner [\![A]\!]\ \Gamma^* \urcorner : s \wedge\ \vdash_n \ulcorner [\![B]\!]\ \Gamma^* \urcorner : s \wedge \downarrow_n \overline{[\![A]\!]} = \downarrow_n \overline{[\![B]\!]}.$$

Preuve. Par la correspondance entre PTS standards et PTS hors contexte et par la correction et la complétude de la normalisation par évaluation. □

La relation de typage est dirigée par la syntaxe puisqu'il existe au plus une règle pour chaque forme syntaxique. On peut donc la lire comme une fonction d'un terme statique vers un appariement d'un type statique et un type dynamique. L'écrire comme une fonction en HASKELL donne le code de la figure 4.9.

```
axiom :: Sort -> Sort
rule  :: Sort -> Sort -> Sort

data STerm = SSort Sort
           | SCast Int STerm
           | SLam STerm (STerm -> STerm)
           | SPi STerm (STerm -> STerm)
           | SApp STerm STerm
           | (:*:) { stat :: STerm, dyn :: DTerm }

data DTerm = DSort Sort
           | DVar Int
           | DLam (DTerm -> DTerm)
           | DPi DTerm (DTerm -> DTerm)
           | DApp DTerm DTerm

of :: Int -> STerm -> STerm
of n (SCast m (v :*: v')) | Sort s <- of m v = v :*: v'
of n (SSort s) = SSort (axiom s) :*: DSort (axiom s)
of n (SLam (v :*: v') f) | let spi = SPi v (\x -> of n (stat (f x)))
                         | let dpi = DPi v (\x -> dyn (of n (stat (f x))))
                         | SSort s <- stat (of n spi)
                         = spi :*: dpi
of n (SPi (v :*: v') f) | SSort s1 <- stat (of n v)
                        | SSort s2 <-
                          stat (of (n+1) (f (SCast n (v :*: v') :*: DVar n)))
                        = rule s1 s2
of n (SApp (v :*: v') (w :*: w')) | SPi (u1 :*: u1') f <- stat (of n v)
                                  | u2 :*: u2' <- of n w
                                  | reify n u1' == reify n u2'
                                  = f (u2 :*: u2')
of n (Let (v :*: v') f) = of (n + 1) (stat (f (SCast n (of n v) :*: DVar n)))
of n _ = error "Type error."
```

Figure 4.9 Implémentation en HASKELL d'un algorithme de typage pour un PTS caractérisé par les fonctions `axiom` et `rule`.

Avec la restriction des termes en forme séquentialisée, la taille de la traduction est maîtrisée et nous évitons ainsi des coercions à la demande des formes statiques vers des formes dynamiques. La transformation des termes en forme monadique est une transformation standard et bien étudiée. Nous considérons que la pleine généralité du λ-calcul est un luxe offert à l'utilisateur, mais qui n'a pas sa place dans un noyau de vérification de preuves petit et digne de confiance. Séquentialiser les termes de l'utilisateur codifie dans le terme lui-même le choix de la stratégie d'évaluation, puisque la séquence de réductions durant l'évaluation de termes en forme monadique est invariante quelque soit la stratégie d'évaluation du métalangage. Éliminer en cela le non-déterminisme de l'évaluation peut-être bénéfique pour d'autres analyses et optimisations sur les preuves.

Nous aurions pu choisir une autre transformation nommant les termes intermédaire, telle que la transformation en forme CPS. Mais cette traduction change le type des termes. Barthe et al. (1999) présente une traduction en style CPS pour une large classe de PTS comprenant notamment tous les systèmes du cube de Barendregt, mais une traduction en style CPS pour tous les PTS n'est, à ma connaissance, pas connue. Kennedy (2007) observe que la forme A-normale n'est pas close par réduction, que la renormalisation nécessaire après réduction peut s'avérer très coûteuse. Employer la forme CPS dans un compilateur peut donc être plus avantageux que la forme A-normale. Nous remarquons que ces problèmes ne sont pas pertinents pour les besoins de ce chapitre, où nous souhaitons simplement contrôler la taille des termes statiques lors de leur appariement avec leur forme dynamique. Le choix d'une forme A-normale est donc approprié, d'autant plus que la forme A-normale peut être vue comme une première étape vers la forme CPS (Danvy, 1991), si cette dernière s'avère plus judicieuse en aval.

4.8 Typage de clôtures

La transformation en forme A-normale dans la section précédente visait à éviter une explosion exponential de la taille des termes à typer, mais il existe toujours des termes source M pour lesquels l'appariement donne des termes cibles dont la taille est une fonction supralinéaire de la taille de M.

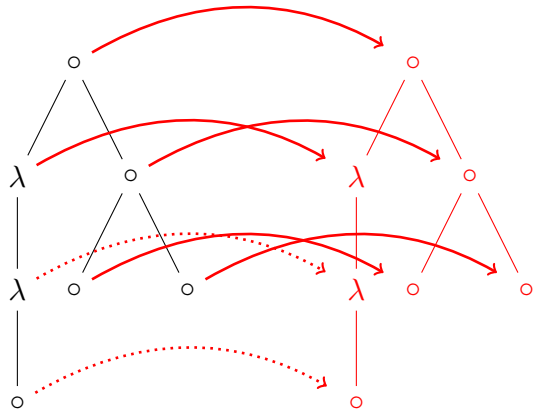


Figure 4.10 Il est impossible de partager la forme dynamique de sous-termes sous les abstractions.

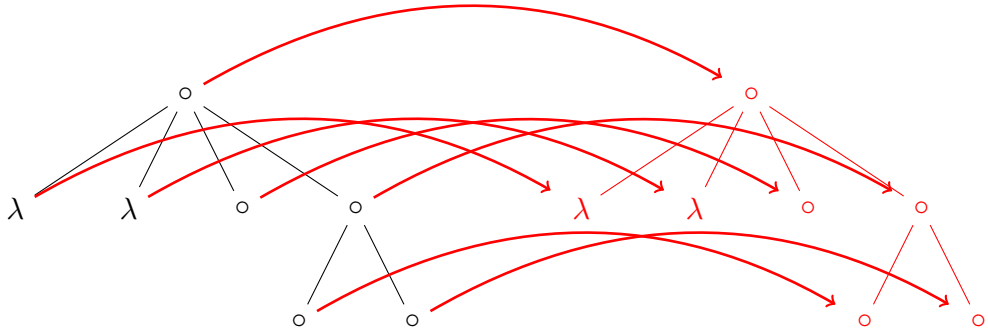


Figure 4.11 Après conversion de clôture, il devient possible de hisser tous les sous-termes apparaissant sous une abstraction à la racine sans risque d'extrusion de portée ou de capture de variables.

Exemple 4.61 Considérons le λ^{pts}-terme suivant

$$\lambda f\ :\mathsf{nat}\to\mathsf{nat}.\ f\ \ (\lambda x:\mathsf{nat}.\ \mathsf{plus}\ x\ x)$$

dans le contexte

$$\mathsf{nat}:\mathbf{Type},\mathsf{plus}:\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}$$

Après application des réductions commutatives pour obtenir une forme A-normale et simplification, la traduction du λ^{pts}-terme ci-dessus donne le $\lambda^{\mathsf{pts}}_{bh}$-terme apparié suivant :

$$\begin{aligned}
&\mathbf{let}\ x_1 \Leftarrow \langle \Pi\ \hat{\mathsf{nat}}\ (\overline{\lambda}y.\ \hat{\mathsf{nat}}), \mathsf{Pi}\ \check{\mathsf{nat}}\ (\overline{\lambda}y.\ \check{\mathsf{nat}})\rangle\ \mathbf{in}\\
&\langle \lambda\ \hat{x_1}\ (\overline{\lambda}f.\ \mathbf{let}\ x_2 \Leftarrow \langle \lambda\ \hat{\mathsf{nat}}\ (\overline{\lambda}x.\ \mathbf{let}\ x_3 \Leftarrow \langle \hat{\mathsf{plus}}\cdot\hat{x}, \underline{\mathsf{app}\ \check{\mathsf{plus}}\ \check{x}}\rangle\ \mathbf{in}\ \langle \hat{x_3}\cdot\hat{x}, \mathsf{app}\ \check{x_3}\ \check{x}\rangle),\\
&\qquad\qquad \mathsf{Lam}\ (\overline{\lambda}x.\ \underline{\mathsf{app}\ (\mathsf{app}\ \check{\mathsf{plus}}\ x)\ x})\rangle\ \mathbf{in}\\
&\qquad \langle \hat{f}\cdot\hat{x_2}, \mathsf{app}\ \check{f}\ \check{x_2}\rangle),\\
&\mathsf{Lam}\ (\overline{\lambda}f.\ \mathsf{app}\ f\ (\mathsf{Lam}\ (\overline{\lambda}x.\ \underline{\mathsf{app}\ (\mathsf{app}\ \check{\mathsf{plus}}\ x)\ x})))\rangle
\end{aligned}$$

L'exemple 4.61 montrent que l'appariement tel qu'opéré dans la définition 4.58 entraîne la duplication de certains sous-termes (soulignés) dans le cas des abstractions et des produits dépendants. Tel qu'illustré dans la figure 4.10, un appariement optimal crée une seule représentation statique et une seule représentation dynamique pour tout le terme, puis associe chaque sous-terme deux-à-deux. L'interprétation $[\![\cdot]\!].$ ne partage pas les formes dynamiques sous les abstractions, puisqu'il est impossible de se référer au corps d'une abstraction à l'extérieur de celle-ci. Une abstraction dynamique est en effet une fonction du métalangage et donc une boîte noire que l'on ne peut inspecter.

Il nous faut opérer une conversion de clôture sur le terme source (figure 4.11) avant d'appliquer l'interpétation $[\![\cdot]\!].$, ce qui nous permet de hisser toutes les abstractions à la racine du terme et ainsi permettre une plus grand partage des formes dynamiques et de ce fait éviter le facteur exponentiel dans la taille de l'interprétation.

Le but d'une conversion de clôture est d'obtenir un terme où le corps de toutes les abstractions sont tous clos (voir chapitre 1, section 1.6.2). Un terme de cette forme est obtenu par l'introduction de nouvelles abstractions. Or dans le cas de termes typés d'un PTS, il nous faut faire attention aux nouvelles abstractions que l'on introduit : tous les PTS ne sont pas clos par conversion de clôture.

Exemple 4.62 Soit M le terme suivant du $\lambda\Pi$-calcul en forme A-normale, dans un contexte où $\mathsf{nat} : \mathbf{Type}$:

$$\mathbf{let}\ x_1 \Leftarrow (\lambda z_1 : \mathsf{nat}.\ \mathbf{let}\ x_2 \Leftarrow \lambda z_2 : \mathsf{nat}.\ z_2\ \mathbf{in}\ x_2)\ \mathbf{in}\ x_1$$

Après conversion de clôture, on obtient le terme suivant :

$$\mathbf{let}\ x_1 \Leftarrow (\lambda z_1 : \mathsf{nat}.\ \mathbf{let}\ x_2 \Leftarrow \lambda y_1 : \mathbf{Type}.\ \lambda z_2 : \mathsf{y}_1.\ z_2\ \mathbf{in}\ x_2\ \mathsf{nat})\ \mathbf{in}\ x_1$$

Ce terme n'est pas un terme bien typé du $\lambda\Pi$-calcul.

Nous devons donc modifier la conversion de clôture pour qu'elle n'introduise pas de nouvelles abstractions qui compromettraient typabilité du résultat. La solution que nous employons est de paramétrer la conversion de clôture par un contexte global Σ. Pour les besoins de cette conversion de clôture modifiée, une variable n'est pas considérée comme apparaîssant libre dans un terme si celle-ci appartient au domaine du contexte global Σ.

Définition 4.63 (Conversion de clôture) Nous définissons la conversion de clôture de termes en forme A-normale comme la fonction suivante, où ρ associe un terme à un nom et Σ est un contexte :

$$
\begin{aligned}
\mathcal{C}_\Sigma(x)_\rho &= \rho(x) \\
\mathcal{C}_\Sigma(\lambda x.\, M)_\rho &= \lambda x.\, \mathcal{C}_\Sigma(M)_{\rho[x\mapsto x]} \\
\mathcal{C}_\Sigma(x\ y)_\rho &= \mathcal{C}_\Sigma(x)_\rho\ \mathcal{C}_\Sigma(y)_\rho \\
\mathcal{C}_\Sigma(\mathbf{let}\ x \Leftarrow N\ \mathbf{in}\ M)_\rho &= \mathbf{let}\ x \Leftarrow \lambda y_1.\cdots\lambda y_n.\, \mathcal{C}_\Sigma(N)\ \mathbf{in}\ \mathcal{C}_\Sigma(M)_{\rho'} \\
&\qquad \text{où } \mathcal{FV}(N) \setminus \mathsf{dom}(\Sigma) = y_1, \ldots, y_n \\
&\qquad \text{et } \rho' = \rho[x \mapsto x\ y_1\ \ldots\ y_n]
\end{aligned}
$$

Il est toujours possible que la conversion de clôture produise un résultat mal typé ; il faut donc choisir Σ de façon adéquat pour garantir la typabilité du résulat. Dans la pratique, on peut choisir pour Σ le contexte global, parfois appelé *signature*, dans lequel baignent les termes que l'on vérifie. Dans le $\lambda\Pi$-calcul, par exemple, les variables dont le type est de sorte $\mathbf{Kind}$ sont forcément des constantes, introduites dans ce contexte global, puisqu'il n'est pas possible d'abstraire sur une variable dont le type est de sorte $\mathbf{Kind}$. Par ailleurs, omettre d'abstraire sur les constantes ne compromet en rien l'utilité de la conversion de clôture, puisque leur portée est globale et donc hisser une constante (voir chapitre 1, section 1.6.2) ne risque pas de provoquer un extrusion de portée.

Exemple 4.64 Le terme source de l'exemple 4.61 devient, après transformation en forme A-normale et conversion de clôture,

$$
\begin{aligned}
&\mathbf{let}\ x_1 \Leftarrow \mathsf{nat} \to \mathsf{nat}\ \mathbf{in} \\
&\lambda f : x_1.\ \mathbf{let}\ x_2 \Leftarrow (\lambda x : \mathsf{nat}.\ \mathbf{let}\ x_3 \Leftarrow \mathsf{plus}\ x\ \mathbf{in}\ x_3\ x)\ \mathbf{in}\ f\ x_2
\end{aligned}
$$

Après hissage de toutes les abstractions, on obtient :

$$
\begin{aligned}
&\mathbf{let}\ x_1 \Leftarrow \mathsf{nat} \to \mathsf{nat}\ \mathbf{in} \\
&\mathbf{let}\ x_2 \Leftarrow (\lambda x : \mathsf{nat}.\ \mathbf{let}\ x_3 \Leftarrow \mathsf{plus}\ x\ \mathbf{in}\ x_3\ x)\ \mathbf{in} \\
&\lambda f : x_1.\ f\ x_2
\end{aligned}
$$

Son interprétation dans un modèle où le terme est apparié avec sa forme dynamique est

$$
\begin{aligned}
&\mathbf{let}\ x_1 \Leftarrow \langle \Pi\ \hat{\mathsf{nat}}\ (\overline{\lambda} y.\ \hat{\mathsf{nat}}), \mathsf{Pi}\ \check{\mathsf{nat}}\ (\overline{\lambda} y.\ \check{\mathsf{nat}})\rangle\ \mathbf{in} \\
&\mathbf{let}\ x_2 \Leftarrow \langle \lambda\ \hat{\mathsf{nat}}\ (\overline{\lambda} x.\ \mathbf{let}\ x_3 \Leftarrow \langle \hat{\mathsf{plus}} \cdot \hat{x}, \underline{\mathsf{app}\ \check{\mathsf{plus}}}\ \check{x}\rangle\ \mathbf{in}\ \langle \hat{x_3} \cdot \hat{x}, \mathsf{app}\ \check{x_3}\ \check{x}\rangle), \\
&\qquad\qquad \mathsf{Lam}\ (\overline{\lambda} x.\ \mathsf{app}\ (\underline{\mathsf{app}\ \check{\mathsf{plus}}}\ x)\ x)\rangle\ \mathbf{in} \\
&\langle \lambda\ \hat{x_1}\ (\overline{\lambda} f.\ \langle \hat{f} \cdot \hat{x_2}, \mathsf{app}\ \check{f}\ \check{x_2}\rangle),\ \mathsf{Lam}\ (\overline{\lambda} f.\ \mathsf{app}\ f\ \check{x_2})\rangle
\end{aligned}
$$

4.9 Des petites aux grandes boîtes

Nous présentons dans cette section une généralisation naturelle du mécanisme de boîte pour aider le partage de sous-structure dans les termes.

La typabilité d'un terme du λ^{st}-calcul est décidable pour des termes à la Church (c'est aussi le cas pour les termes à la Curry). Pour les termes *clos* d'un PTS $\mathcal{P} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$, la typabilité n'est en général décidable que pour les termes à la Church. Dans ce cas, la conjonction des trois conditions suivantes est suffisante pour la décidabilité du typage (van Benthem Jutting, 1993) :

1. le PTS est fonctionnel,
2. le PTS est fortement normalisant,
3. $\mathcal{S}$ est de taille finie.

La typabilité des termes ouverts d'un PTS peut néanmoins être elle aussi décidable pour certaines sous-classes de termes de ce PTS (par exemple les termes du système F dont le type est de rang 1, correspondant au polymorphisme à la ML (Milner, 1978), ou de rang 2 (Kfoury et Wells, 1994)). Quoi qu'il en soit, les algorithmes décidant de la typabilité quand cela est possible reposent souvent sur des algorithmes d'unification pour propager les contraintes inférées lors du parcours d'un sous-terme dans un autre sous-terme. Mais si l'on connaît le type des variables libres d'un terme, annoter ces variables avec leur type permet à un terme de porter en lui toutes les informations nécessaires pour décider du type de ce terme sans l'aide d'algorithmes d'unification, ni même d'environnement de typage.

Il peut être intéressant d'aller plus loin, en permettant non seulement des annotations sur les variables mais aussi sur des sous-termes arbitraires.

Définition 4.65 (λ_B^{pts}-calcul) Soit un PTS $\mathcal{P} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$. La syntaxe des λ_B^{pts}-termes est identique à celle de λ_b^{pts}-termes, à ceci près que les annotations peuvent se faire sur des termes arbitraires :

$$\begin{array}{rcll} \mathcal{S} & \ni & s, s_n & \\ \mathsf{Term_B} & \ni & M, N, A, B & ::= \quad s \mid x \mid \lambda x : A.\ M \mid \Pi x : A.\ B \mid M\ N \mid [M : A] \end{array}$$

Nous définissons de manière analogue les termes du λ_B^{st}-calcul.

Nous parlons alors de *grandes boîtes* pour les annotations de types dans les termes, par opposition aux boîtes ne pouvant contenir que des variables, dont on pourrait dire qu'elles sont petites. Il est alors possible de vérifier la typabilité d'un terme obtenu par combinaison de deux sous-termes annotés $[M_1 : A]$ et $[M_2 : B]$ sans reparcourir M_1 et M_2.

4.9.1 Principes définitionnels

La possibilité de grandes boîtes est particulièrement intéressante pour l'architecture LCF. Les implémentations de HOL se passent de grandes boîtes car celles-ci comptent sur une extension de la logique originelle avec des principes définitionnels pour contrôler la taille des termes et ainsi limiter le coût de l'inférence de types nécessaire durant la combinaison de deux termes.

Ces principes définitionnels permettent d'énoncer, de façon systématique et sans compromettre la cohérence de la logique, des axiomes postulant une égalité entre un nouveau nom c, appelé *constante*, et un terme donné M. L'utilisateur peut alors écrire cette constante en lieu et place de M partout où celui-ci apparaît. Les principes définitionnels sont des outils cruciaux pour contrôler la taille des termes, des formules et de leurs démonstrations. Les constantes, comme les variables, ayant toujours un type manifeste, élaguer un terme en remplaçant nombre de ses branches par des constantes est un moyen efficace d'assurer la bonne performance de l'opération de combinaison de ce terme avec un autre.

Un assistant de preuves se doit dans la pratique de fournir un mécanisme de définitions à l'utilisateur. Vues à travers la lentille de la correspondance de Curry-Howard, les définitions sont autant de lemmes dotés de noms, que l'on pourra invoquer une ou plusieurs fois dans le terme de preuve démontrant le théorème final. Pour autant, nous considérons ici que les définitions ne sont que du sucre syntaxique indépendant de la représentation des termes dans le noyau, qui ne doit nullement s'immiscer dans la théorie implémentée par celui-ci. En effet, les axiomes définitionnels[37] demandent à être validés (Gordon et Melham, 1993chapitre 16) et la préservation de la normalisation forte (et donc de la cohérence) par ajout de constructions de définitions aux termes des PTS reste, à notre connaissance, un problème ouvert (bien qu'il soit résolu par l'affirmative pour une large classe de PTS (Severi et Poll, 1994)).

En l'absence de définitions, rien n'empêche de représenter dans le noyau un terme sous la forme d'un graphe, tel qu'illustré dans la figure 4.12. Dans un langage fonctionnel, le partage de sous-structures n'est néanmoins pas observable[38]. Aussi, il est important que les noeuds partagés soient annotés par leur type pour éviter que celui-ci ne soit recalculé, à défaut de mécanismes plus élaborés (Gill, 2009). Les grandes boîtes servent alors à annoter le type des noeuds partagés.

[37] Les principes définitionnels des premières implémentations de HOL étaient intuitivement conservatifs, mais Mark Saaltink et Roger Jones démontrèrent indépendemment que certaines définitions permises pouvaient compromettre la cohérence du système (Gordon, 2000).

[38] Dans la mesure où celles-ci sont référentiellement transparentes.

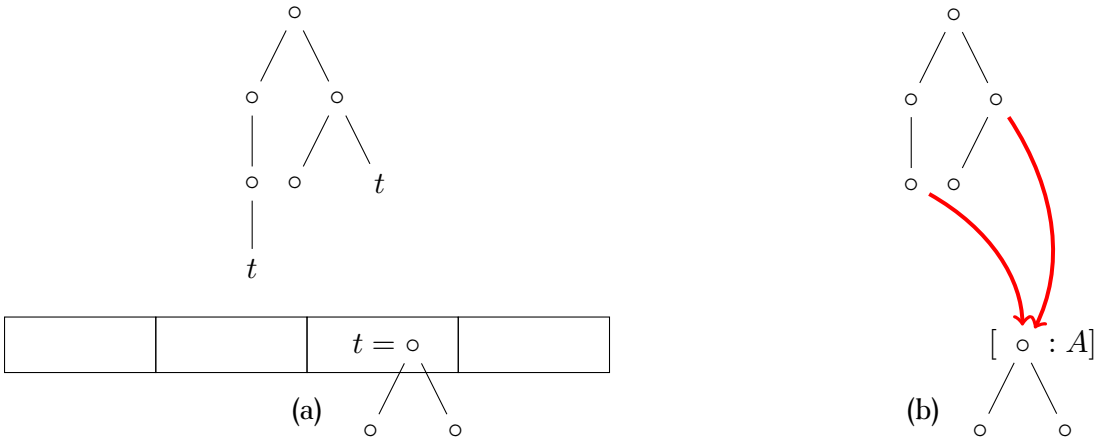


Figure 4.12 Représentation d'un terme (a) sous un contexte global ou (b) avec partage.

4.9.2 Élimination du contexte global

L'architecture LCF mélange vérification des types avec construction des termes, alors que les descendants de AUTOMATH séparent les deux phases. Cette séparation induit que certains termes sont certifiés bien typés, alors que d'autres termes ont un statut inconnu. Ces assistants maintiennent souvent un registre global de termes bien typés indexés par des noms — autrement dit un registre de définitions (représenté par des cases dans la figure 4.12(a)), aussi appelé *contexte global de typage*. À chaque nouvelle définition $x := M$ de l'utilisateur correspond une mutation du contexte global pour y ajouter x, M et le type de M après vérification que M est typable. Ce contexte global sera soit filé à travers nombre des fonctions de l'interface du noyau, ou alors sera mis à jour au fil des besoins par effet de bord. Dans COQ, ajouter une nouvelle définition au contexte global est la prérogative de

```
val add_constant : constant -> constant_body -> env -> env
```

L'architecture LCF admet quant à elle une implémentation purement fonctionnelle et sans manipulation de contexte global explicite. Une variante dite « stateless » de HOL est d'ailleurs implémentée ainsi (Wiedijk, 2009), en utilisant une variante des grandes boîtes. Les termes ne sont plus représentés comme des arbres dont les feuilles sont des variables ou des constantes, mais comme des graphes où les constantes pointent vers leur définition (voir la figure 4.12(b)), de sorte que les feuilles ne sont plus que des variables.

4.9.3 Commutation de la β-réduction sous les grandes boîtes

Changer la grammaire des annotations induit une notion légèrement différente de substitution.

Définition 4.66 Nous étendons les égalités définissant la substitution dans la définition 1.14 avec l'égalité suivante :

$$\{N/x\}[M : A] = [\{N/x\}M : \{N/x\}A]$$

La conséquence majeure de cette définition est que même certains des λ_b^{st}-termes typables et *clos* ne le sont pas dans un système avec contextes, comme le montre l'exemple suivant :

$$\lambda x : \tau\,.\,[x : \tau \to \tau]\; x$$

Bien que ce terme soit typable dans le système de types hors contexte, ainsi que le montre la dérivation suivante,

$$\dfrac{\dfrac{\dfrac{}{\vdash [[x : \tau] : \tau \to \tau] : \tau \to \tau} \quad \dfrac{}{\vdash [x : \tau] : \tau}}{\vdash [[x : \tau] : \tau \to \tau]\,[x : \tau] : \tau}}{\vdash \lambda x : \tau\,.\,[x : \tau \to \tau]\; x : \tau \to \tau}$$

le terme $\lambda x : \tau\,.\; x\; x$, obtenu par effacement des annotations, n'est pas typable dans le λ-calcul simplement typé. Auparavant, ces termes « exotiques » étaient l'apanage des termes ouverts. Cette observation ne met pas pour autant en péril les théorèmes de complétude et de correction énoncés précédemment.

Une autre conséquence de la présence d'annotations sur des termes quelconques est que ces annotations peuvent maintenant bloquer la β-réduction. Il faut donc étendre la réduction des λ-termes.

Définition 4.67 La relation de réduction $\longrightarrow_{\beta B}$ est l'union de $(\longrightarrow_\beta)$ et de la règle suivante :

$$[(\lambda x : \tau\,.\,M) : \sigma]\; N \longrightarrow_B [\{N/x\}M : \sigma]$$

Il faut alors réétablir quelques notions usuelles de cette relation de réduction étendue, notamment la propriété de Church-Rosser (CR), la propriété de préservation de la normalisation forte (PSN) et la réduction du sujet (SR). Dans cette thèse, nous laissons ces propriétés à l'état de conjecture, car nous n'identifions pas ici les termes étendus avec des boîtes modulo β-équivalence. En effet, dans les systèmes de types précédants, les langages de termes avec boîtes sont tous des langages statiques, qui ne servent pour le typage. Nous n'identifions modulo β-équivalence que les termes dynamiques, qui eux ne contiennent pas de boîtes.

4.10 Conclusion

Le chapitre 3 montrait comment calculer efficacement avec des termes de preuves. Nous avions introduit pour cela une représentation des preuves commes des programmes, c'est à dire

des structures d'ordre supérieur car nous représentions les abstractions par les abstractions du métalangage. Nous avons tenté dans ce chapitre de modifier l'interprétation des termes donné dans le chapitre 3 pour obtenir encore une fois des structures d'ordre supérieur pour représenter les preuves. Mais cette fois-ci pour vérifier leur type plutôt que de calculer avec. Nous avons hérité ainsi de la même facilité à faire des substitutions, nécessaires durant le typage des applications dans une théorie des types dépendante. Cette représentation à l'ordre supérieur nous a amené à se débarasser du contexte de typage, puisque nous pouvions tout aussi bien réutiliser l'environnement des clôtures sous-jacentes servant à représenter notre interprétation des termes de preuve. Nous arrivons ainsi à des systèmes de types purs où les contextes sont absents des jugements de typage, qui sont corrects et complets par rapport aux systèmes usuels avec contextes. Enfin, nous avons montré comment arriver à un système de types rendant compte de la double interprétation des termes nécessaire pour vérifier ces termes et calculer avec, que nous avons pu écrire directement sous forme d'une fonction dans un langage de programmation. Nous avons motivé la nécessité de transformer les termes dans une forme plus simple où toutes les formes intermédiaires sont nommées, afin de limiter la duplication du code.

Les preuves de correspondance entre systèmes avec et sans contextes sont entièrement formalisées en Coq pour le λ-calcul simplement typé. La preuve de complétude pour le cas général des PTS est aussi formalisée en Coq. La preuve de correction n'est que partiellement formalisée. Nous travaillons actuellement sur la transformation en forme A-normale pour arriver à une formalisation bout à bout des résultats présentés dans ce chapitre, à partir de laquelle il sera possible d'extraire un algorithme de typage efficace et certifié.

chapitre 5
Usages et encodages

Nous rassemblons dans ce chapitre les briques élémentaires construites dans les chapitres précédents pour présenter Dedukti, un vérificateur de preuves pour le $\lambda\Pi$-calcul modulo. Nous donnons également une traduction des preuves du CCI restreint à un univers vers le $\lambda\Pi$-calcul modulo. Nous avons ainsi un exemple d'utilisation de Dedukti comme vérificateur d'un sous-ensemble de preuves du système Coq.

5.1 Compilation de preuves complètes

Le développement de Dedukti se plie aux contraintes qu'impose le critère de de Bruijn. Le noyau sur lequel repose la correction de tout le système se doit d'être simple afin d'être digne de notre confiance. La simplicité la plus facile est la concision, que nous ne pouvons espérer atteindre qu'en réutilisant des composantes logicielles existantes standards. Nous avons montré dans le chapitre 3 et le chapitre 1 comment réutiliser quelques-unes de ces composantes, d'une part pour calculer sur les preuves et d'autre part pour vérifier ces preuves. Ces deux chapitres reposent sur des traductions de structures au premier ordre vers des structures à l'ordre supérieur. Conceptuellement, nous prenons des programmes en entrée pour fabriquer des programmes en sortie. L'évaluation de ces programmes donne le résultat souhaité. Comme noté dans le chapitre 1, les programmes gagnent à être compilés pour de bien meilleures performances. Nous arrivons donc à l'architecture illustrée dans la figure 5.1.

Le format d'entrée de Dedukti est un fichier `.dk` contenant exactement une unité de compilation, ou *module*. Celui-ci est une liste de déclarations donnant le type d'un certain nombre de constantes. Ces déclarations forment la *signature* du module. Le fichier `.dk` peut également attacher des règles de réécriture à certaines constantes de la signature. Chaque constante est groupée avec les règles de réécriture qui lui sont attachées dans un *paquet*. Chaque paquet est traduit en deux déclarations en Haskell : une déclaration pour la forme dynamique et une déclaration pour la forme statique.

Dedukti agit ainsi comme un traducteur de fichiers `.dk` en fichiers `.hs`, c'est à dire de scripts de preuves en $\lambda\Pi$-modulo vers des modules de programmes fonctionnels, ici dans le langage de programmation Haskell. Ces fichiers sont ensuite compilés (ici par le compilateur GHC) vers des *fichiers objets* contenant du code natif pour chacun des modules. Enfin, ces fichiers objets sont *liés* ensemble et le programme obtenu exécuté, par le script `dkrun`.

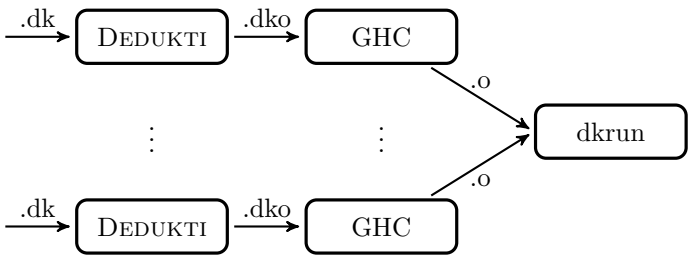


Figure 5.1 Schéma de compilation d'une preuve par DEDUKTI.

Nous trouvons à cette architecture les avantages suivants :

- la simplicité — toute la traduction est faite d'un coup sans avoir à gérer explicitement un état courant que l'on met à jour à la traduction de chaque nouvelle déclaration ou règle de réécriture ;
- la performance — les termes de preuves sont compilés pour calculer avec plus rapidement ;
- la performance encore — compiler toutes les preuves d'un coup permet de s'affranchir du surcroît de bureaucratie nécessaire à des appels répétés au compilateur pour compiler seulement certains morceaux de preuves ponctuellement. Ce dernier point est particulièrement pertinent pour le champs d'application principal de DEDUKTI : un plongement des preuves de théories étrangères en $\lambda\Pi$-modulo induit l'existence de nombreux radicaux supplémentaires (dûs aux règles de réécriture nécessaires à l'encodage) par rapport aux preuves initiales. Les conversions de types sont donc beaucoup plus fréquentes.

Notons cependant que cette simplicité peut avoir un coût. Les temps de compilation peuvent être élevés, en particulier lorsqu'on demande au compilateur d'optimiser le code produit. Ces optimisations bénéficieront principalement les formes dynamiques utiles à la conversion, mais n'auront quasiment aucun impact sur les formes statiques, dont le contenu calculatoire est très faible. Or la traduction de DEDUKTI mélange les formes statiques et dynamiques d'un terme, de sorte que le compilateur optimisera à la fois les formes dynamiques et statiques — ces dernières en pure perte.

Compiler les preuves en un coup est approprié pour un vérificateur de preuves complètes produites par une multitude d'autres systèmes. Dans un usage interactif, les preuves sont souvent construites par raffinement — à savoir en laissant des trous (marqués par des *métavariables*) dans la preuve qui seront remplis au fur et à mesure par l'utilisateur. Compiler à la volée et en continu des preuves partielles en injectant du code à chaque fois qu'un trou dans une preuve est bouchée nous paraît à tout le moins difficile. Nous remarquons cependant qu'il faut distinguer

représentations des preuves manipulables par des tactiques de représentations pour le noyau d'un assistant de preuves. Dans le système Coq, par exemple, la preuve d'une formule du contexte global n'est validée par le noyau qu'une fois éliminées toutes les métavariables dans la preuve. Nous pouvons donc compiler qu'une seule fois le terme de preuve, une fois que celui-ci est complètement élaboré. De même, HOL Light contient une multitude de représentation des termes, certaines permettant des occurrences de métavariables. Mais il n'existe pas de métavariables dans la représentation des termes validés par le noyau de HOL Light.

5.2 Plongement du CCI

Cousineau et Dowek (2007) montrent comment encoder les termes du calcul des constructions dans le $\lambda\Pi$-calcul modulo. Dans cette section, nous montrons comment étendre cet encodage aux types inductifs. Cette extension a été faite avec le concours de Guillaume Burel, auteur d'un outil de traduction[39] des termes de preuves de Coq vers Dedukti. Le calcul des constructions tel que implémenté dans Coq contient trois sortes $\mathbf{Set}$, $\mathbf{Prop}$, $\mathbf{Type}$ où $\mathbf{Set}, \mathbf{Prop} : \mathbf{Type}$, avec sept règles pour le type des produits dépendants en fonction du type du domaine et du codomaine. Ainsi le calcul des constructions a en fait sept produits. Le schéma de la section section 2.5 donne une signature contenant une constante pour chacun des produits et les règles de réécriture suivantes :

$$\epsilon_{Kind}\ (\dot{Type}) \longrightarrow U_{Type}$$
$$\epsilon_{Kind}\ (\dot{Prop}) \longrightarrow U_{Prop}$$
$$\epsilon_{Prop}\ (\dot{\Pi}_{\langle Prop,Prop,Prop\rangle}\ A\ B) \longrightarrow \Pi x : \epsilon_{Prop}\ A.\ \epsilon_{Prop}\ (B\ x)$$
$$\epsilon_{Kind}\ (\dot{\Pi}_{\langle Prop,Kind,Kind\rangle}\ A\ B) \longrightarrow \Pi x : \epsilon_{Prop}\ A.\ \epsilon_{Kind}\ (B\ x)$$
$$\epsilon_{Kind}\ (\dot{\Pi}_{\langle Kind,Prop,Kind\rangle}\ A\ B) \longrightarrow \Pi x : \epsilon_{Kind}\ A.\ \epsilon_{Prop}\ (B\ x)$$
$$\epsilon_{Type}\ (\dot{\Pi}_{\langle Type,Type,Type\rangle}\ A\ B) \longrightarrow \Pi x : \epsilon_{Type}\ A.\ \epsilon_{Type}\ (B\ x)$$
$$\epsilon_{Kind}\ (\dot{\Pi}_{\langle Type,Kind,Kind\rangle}\ A\ B) \longrightarrow \Pi x : \epsilon_{Type}\ A.\ \epsilon_{Kind}\ (B\ x)$$
$$\epsilon_{Kind}\ (\dot{\Pi}_{\langle Kind,Type,Kind\rangle}\ A\ B) \longrightarrow \Pi x : \epsilon_{Kind}\ A.\ \epsilon_{Type}\ (B\ x)$$
$$\epsilon_{Kind}\ (\dot{\Pi}_{\langle Kind,Kind,Kind\rangle}\ A\ B) \longrightarrow \Pi x : \epsilon_{Kind}\ A.\ \epsilon_{Kind}\ (B\ x)$$

Nous étendons la signature pour donner le type de chaque type inductif que l'on veut traduire et le type de chaque constructeur de chaque inductif. Nous assignons un numéro

[39] http://github.com/gburel/coqine

unique à toutes les analyses de cas dans tous les termes que l'on veut traduire et étendons la signature avec une constante pour chaque analyse de cas :

$$\dot{\mathsf{case}}^u : |\Pi y_1 : A_1. \cdots \Pi y_n : A_n.\ I\ y_1 \ldots y_n \to P\ y_1 \ldots y_n|$$

où u est le numéro de l'analyse de cas, $n - l$ est le nombre d'indices de l'inductif I (voir la définition 2.13) et P est le prédicat donnant le type de retour de l'analyse par cas. Enfin, nous supposons que pour chaque point fixe $\mathsf{fix}\, f : A := M$ le nom f identifie de manière unique le point fixe et avons une nouvelle constante pour chaque point fixe[40] :

$$f : |A \to B|$$

où A est le type de l'argument du point fixe et B le type de retour.

Il ne nous reste plus qu'à ajouter des règles de réécriture pour chacune des branches de chacune des analyses par cas et chaque point fixe. Pour l'analyse par cas u, nous ajoutons les règles suivantes :

$$\begin{array}{c}
[y_1 : |A_1|, \ldots, y_n : |A_n|, x_1 : |B_1|, \ldots, x_{m_1} : |B_{m_1}|]\, \dot{\mathsf{case}}^u\ y_1 \ldots y_n\ (C_1\ x_1 \ldots x_{m_1}) \longrightarrow |M_1| \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
[y_1 : |A_1|, \ldots, y_n : |A_n|, x_1 : |B_1|, \ldots, x_{m_k} : |B_{m_k}|]\, \dot{\mathsf{case}}^u\ y_1 \ldots y_n\ (C_n\ x_1 \ldots x_{m_k}) \longrightarrow |M_k|
\end{array}$$

où m_i est l'arité du constructeur C_i, $y_1, \ldots, y_l$, sont les paramètres et $y_l, \ldots, y_n$ les indices du type inductif I du terme analysé, et $M_1, \ldots, M_k$ sont le corps de chacune des branches.

Pour caractériser le comportement calculatoire des points fixes, il nous faut rajouter des réécriture s'y rapportant. Notons que la réduction de ces derniers se fait d'après la règle $(unfold)$, et que celle-ci ne déplie le point fixe que si l'argument du point fixe a un constructeur en position de tête. Puisque nous avons choisi un plongement superficiel plutôt qu'un plongement profond des termes du CCI dans le $\lambda\Pi$-calcul modulo, il n'est pas possible de détecter directement si la tête d'un terme est un constructeur ou si c'est plutôt une variable. On ne peut donc écrire qu'une seule règle de réécriture exprimant l'idée que l'application d'un point fixe à un terme M est un radical si et seulement si M n'est pas une variable. Il nous faut opter pour l'approche duale qui consiste à donner une règle de réécriture pour chaque constructeur possible étant donné le type de M.

Mais les dépendances dans les types compliquent l'affaire : de manière générale, un point fixe unaire prend la forme $\mathbf{fix}\, f : I\ N_1\ \ldots\ N_n := M$ où l et $n - l$ sont le nombre de

[40] Comme dans le chapitre 3, nous nous limitons ici aux points fixes unaires par souci de clarté.

paramètres et le nombre d'indices (voir la définition 2.13) du type inductif I, respectivement. Ainsi, une règle de la forme

$$[\ldots, x_1 : |B_1|, \ldots, x_m : |B_m|]\ f\ (C_i\ x_1\ \ldots\ x_m) \longrightarrow |M|\ (C_i\ x_1\ \ldots\ x_m)$$

n'est pas bien typée si les indices dans le type du constructeur C_i ne sont pas tous convertibles deux à deux avec $N_l, \ldots, N_n$. Il nous faut introduire une nouvelle constante, $\dot{f_{red}}$, qui joue un rôle analogue aux constantes $\dot{\mathsf{case}}^u$ pour les analyses de cas :

$$\dot{f_{red}} : |\Pi y_1 : A_1. \cdots \Pi y_n : A_n.\ I\ y_1\ \ldots\ y_n \to B|$$

et ajouter les règles de réécriture suivantes :

$$[x : I\ |N_1|\ \ldots\ |N_n|]\ f\ x \longrightarrow \dot{f_{red}}\ |N_1|\ \ldots\ |N_n|\ x$$

$$[y_1 : |A_1|, \ldots, y_n : |A_n|, x_1 : |B_1|, \ldots, x_{m_1} : |B_{m_1}|]\ f_{red}\ y_1\ \ldots\ y_n\ (C_i\ x_1\ \ldots\ x_{m_1}) \longrightarrow |M|$$

$$\vdots \qquad\qquad \vdots$$

$$[y_1 : |A_1|, \ldots, y_n : |A_n|, x_1 : |B_1|, \ldots, x_{m_k} : |B_{m_k}|]\ f_{red}\ y_1 \ldots\ y_n\ (C_i\ x_1\ \ldots\ x_{m_k}) \longrightarrow |M|$$

La traduction $|\cdot|$ est étendue aux analyses de cas comme suit :

$$|\mathbf{case}_I\ (M_S, M_P, N_1| \ldots |N_k)| = \dot{\mathsf{case}}^w\ |M_l|\ \ldots\ |M_n|\ |M_S|$$

où w est un numéro unique et $M_l, \ldots, M_n$ sont les indices du type de M_S. La traduction $|\cdot|$ est étendue aux points fixes comme suit :

$$|\mathbf{fix}\ f\ : A := M| = f$$

Exemple 5.1 Considérons la fonction `head` de Coq sur les vecteurs de booléens indexés par leur taille définie de la manière suivante :

```
Definition P n := match n with O => unit | S n' => bool end.
Definition head xs :=
  match xs as bvec n return P n with
  | nil => tt
  | cons n x xs => x
  end.
```

Un plongement de ce terme dans dans le $\lambda\Pi$-calcul modulo donnera entre autres les constantes suivantes dans la signature :

$$\mathsf{head} : |\Pi n : \mathsf{nat}.\ \mathsf{bvec}\ (\mathsf{S}\ n) \to \mathsf{bool}|$$
$$\dot{\mathsf{case}}^0 : |\Pi n : \mathsf{nat}.\ \mathsf{bvec}\ n \to P\ n|$$

Nous avons entre autres règles de réécriture :

$$[n : \mathsf{nat}, xs : \mathsf{bvec}\ (\mathsf{S}\ n)]\ \mathsf{head}\ n\ xs \longrightarrow \dot{\mathsf{case}}^0\ (\mathsf{S}\ n)\ xs$$
$$[n : \mathsf{nat}]\ \dot{\mathsf{case}}^0\ n\ \mathsf{nil} \longrightarrow \mathsf{tt}$$
$$[n : \mathsf{nat}, x : \mathsf{bool}, xs : \mathsf{bvar}\ n]\ \dot{\mathsf{case}}^0\ n\ (\mathsf{cons}\ n\ x\ xs) \longrightarrow x$$

5.2.1 Définitions globales et locales

Le CCI tel qu'implémenté dans Coq permet des définitions locales grâces aux constructions **let** ... **in** ainsi que des définitions de constantes dans le contexte global. Ces fonctionnalités sont essentielles pour contenir la taille des preuves, limiter l'usage de ressources mémoire et accélérer la conversion de types (Barras, 2000). Pour autant, le $\lambda\Pi$-calcul modulo en est dépourvu. Ces mécanismes seraient redondants, puisqu'il est possible de les émuler avec des règles de réécriture.

Une définition globale $x : A := M$ donne lieu à l'ajout d'une constante $x : |A|$ dans la signature et une règle de réécriture

$$[]\ x \longrightarrow |M|$$

simulant la règle (δ) du CCI. Les définitions locales peuvent quant à elles soit être simulées par des abstractions ou remontées jusqu'au niveau racine par un algorithme de « let floating » (Peyton Jones et al., 1996), devenant ainsi des définitions globales qui peuvent être ajoutées à la signature.

5.2.2 Constantes opaques

De nombreux termes de preuves n'ont comme seul intérêt d'exister, sans qu'il soit utile de calculer avec. Par exemple, le contenu calculatoire de toutes les preuves d'égalité n'est pas très intéressant : elles sont toutes convertibles à la fonction identité. Coq permet donc de rendre certaines définitions globales *opaques*, au sens où elles se comportent comme des axiomes durant la conversion. Ce mécanisme permet d'éviter de nombreux calculs inutiles durant la vérification de preuves.

Encore une fois, nous faisons l'économie de ce mécanisme dans Dedukti. D'une part, de nombreux lemmes n'ont intérêt à être opaques que ponctuellement. Calculer avec une preuve

d'un lemme ou non devrait être laissé au choix de l'utilisateur du lemme. Idéalement, ce choix peut être modélisé comme une stratégie d'évaluation. Mais dans Dedukti, nous n'avons que peu de contrôle sur la stratégie d'évaluation du langage hôte vers lequel les preuves sont compilées.

Nous empruntons donc l'idée de verrouillage/déverrouillage local d'une constante afin d'éviter ou non de calculer avec en fonction des besoins à Ssreflect (Gonthier et Mahboubi, 2008), et nous l'adaptons au $\lambda\Pi$-calcul modulo. Étant donné un objet de type A que nous voulons opacifier, nous introduisons la constante

$$\mathsf{lock}_A : A \to A$$

avec l'axiome suivant :

$$\mathsf{locked}_A : \mathsf{lock}_A\ A = A.$$

Une constante verrouillée $\mathsf{lock}_A\ c$ ne participe plus, désormais, à la β-réduction, car elle n'est pas convertible à c. Mais étant donné un terme $\mathsf{lock}_A\ c$, nous pouvons le réécrire avec l'axiome locked_A pour récupérer c.

Conclusion

Cette thèse s'était donné comme but d'implémenter un vérificateur complet, de confiance et efficace pour le $\lambda\Pi$-calcul modulo. Dans notre approche, nous avons placé la compilation des termes de preuves de ce calcul au centre du dispositif, en traduisant les preuves vers des programmes lorsqu'on veut calculer avec, et en les traduisant vers de la syntaxe abstraite d'ordre supérieur lorsqu'on veut vérifier ces preuves. La vérification dépend de la capacité à calculer sur les preuves, et le calcul ne peut se faire de façon sûre (sans boucler à l'infini) que si les preuves sont vérifiées au préalable. Ainsi dans un système complet de vérification de preuves tel que Dedukti, il est nécessaire de garder les deux représentations à portée de main.

Notre présentation de ces deux représentations a mis l'accent sur le fait qu'elles étaient très voisines. En effet, il s'agit dans un cas comme dans l'autre d'interpréter les formes syntaxiques du $\lambda\Pi$-calcul modulo vers les formes syntaxiques correspondantes dans un langage hôte, tel que Haskell ou OCaml. La seul différence entre l'interprétation du calcul et l'interprétation de la vérification réside dans le traitement de l'application : il faut traduire le terme $M\ N$ en $\mathsf{app}\ [\![M]\!]\ [\![N]\!]$ pour calculer avec, mais en $\mathsf{App}\ [\![M]\!]\ [\![N]\!]$ pour vérifier le typage de celui-ci avant de calculer avec.

Nous retrouvons cette même dualité entre encodage dans le langage hôte d'un terme et encodage de sa sémantique dans la façon dont on manipule ces termes. Pour connaître la forme normale d'une abstraction, il faut transformer une variable (liée) en un paramètre (une variable libre). Nous ne connaissons pas le nom de la variable liée — nous n'y avons pas accès et ce nom n'est pas pertinent. Il faut donc lui donner un nom, que l'on aura inventé pour l'occasion, afin d'en faire un paramètre. Une fois instanciée la variable liée avec ce nouveau paramètre, nous obtenons l'accès au corps de l'abstraction.

De même, dans le système de type hors contexte sur les termes à l'ordre supérieur, le type du corps d'une abstraction est donné par une relation de typage reliant le corps de l'abstraction où la variable liée (anonyme) a été instanciée par un paramètre (nommé). À la différence près que le paramètre est ici accompagné de son type.

Dans le sens inverse, abstraire sur un terme consiste à remplacer un paramètre par une variable du métalangage. Le nom de cette variable n'est pas pertinent, aussi nous laissons au métalangage le soin de choisir un nom qui soit frais pour cette variable (ou même d'en faire un simple indice de De Bruijn).

Dans cette lumière, les représentations du chapitre 3 (pour le calcul) et du chapitre 4 (pour le typage) sont toutes deux des instances de représentations « locally nameless ». Dans cette

approche (McKinna et Pollack, 1993, McBride et McKinna, 2004 et Aydemir et al., 2008), les variables liées sont typiquement représentées par des indices de De Bruijn, bien que ce soit là un détail d'implémentation. Ce qui caractérise l'approche est l'existence des deux seules primitives suivantes comme fonctions manipulant les variables liées :

$$\mathsf{abstract} : \mathsf{Name} \to \mathsf{term} \to \mathsf{Scope}$$

$$\mathsf{instantiate} : \mathsf{Term} \to \mathsf{Scope} \to \mathsf{Term}$$

Dedukti est donc une implémentation « locally nameless », où les variablies liées sont des variables du métalangage plutôt que des indices. Un Scope (une portée) est une fonction de type Term → Term. La fonction instantiate est plus facile encore qu'avec des indices : une substitution en HOAS, c'est à dire une simple application.

Cette unité de mécanisme conceptuel permet une implémentation simple et très concise : le nombre de lignes de Dedukti, du parseur au générateur de programme, est de l'ordre de 1300 lignes. Une telle concision est un avantage de taille pour asseoir le confiance que l'on est prêt à accorder à un tel outil. Cette concision est rendue possible par la construction presque systématique d'un modèle de termes approprié pour chaque problème et en choisissant ces modèles tels qu'ils peuvent être implémentés facilement avec des outils logiciels existants.

Mais nous devons également cette concision à la relative simplicité de la tâche à accomplir : vérifier les preuves du $\lambda\Pi$-calcul modulo, qui est un système simple et uniforme faisant l'économie de nombreuses notions primitives d'autres systèmes, tels que les types inductifs, une hiérarchie d'univers ou encore des procédures de décision intégrées à la théorie. Ces notions, souvent essentielles pour le développement de grosses démonstrations, sont reléguées à des surcouches au dessus de Dedukti.

L'intérêt d'une telle approche est la modularité. Dedukti est au centre d'une nébuleuse d'outils futurs permettant à l'utilisateur de faire les choix qui lui conviennent, optimisant taille de la base de confiance versus performance ou facilité d'écriture des preuves. Nous souhaiterions ainsi poursuivre de futurs travaux dans les directions suivantes.

Terminaison et confluence des règles de réécriture La cohérence de la logique de Dedukti dépend crucialement de bonnes propriétés du système de règles de réécriture que l'on utilise pour étendre son expressivité. En particulier, il est nécessaire que ce système de règles soit confluent et fortement normalisant. Nous avons étudié dans cette thèse l'implémentation d'un vérificateur de types. Mais la validité d'une preuve dépend aussi des bonnes propriétés des règles de réécriture servant à les encoder. Ces bonnes propriétés peuvent être vérifiées par des outils dédiés, tels que CiME (Contejean et al., 2003), augmentant ainsi la taille de la base de confiance. CiME peut aussi générer des certificats, que Dedukti pourrait vérifier (Blanqui et al.,

2006 et Contejean et al., 2010), afin d'éviter la nécessité de conférer sa confiance à d'autres outils externes en plus de Dedukti.

Vérification de conditions de bord La validité de preuves dans des théories très souples tels que le calcul des constructions inductives avec points fixes de Coq ne dépend plus exclusivement de la bonne typabilité. Coq doit par exemple vérifier des conditions syntaxique de garde sur les points fixes, de manière à interdire la construction de termes qui bouclent et qui pourraient potentiellement compromettre la cohérence logique du système. Nous avons le choix ici soit de développer un outil spécifique pour vérifier ces conditions, ou encoder ces conditions syntaxiques sous la forme de contraintes de typage. Cody Roux travaille actuellement sur une extension de Dedukti permettant des annotations de taille dans les types pour vérifier la terminaison.

De même, Coq inclue une hiérarchie d'univers dans lesquels placer les termes de preuves. Afin de maximiser la réutilisation de lemmes existants, Coq ne demande pas à l'utilisateur de fixer le niveaux de l'univers de chaque lemme explicitement. Celui-ci assigne un univers à chaque occurrence de la sorte `Type` et génère des contraintes globales sur tous les univers connus. La construction de nouveaux termes provoque l'ajout de nouvelles contraintes à l'ensemble de contraintes existantes, que Coq vérifie à chaque instant comme étant satisfiable.

Nous remarquons que dans la pratique, les développements de preuves Coq n'utilisent que très peu d'univers. Nos investigations préliminaires montrent qu'il est par exemple possible d'assigner des univers à tous les théorèmes et toutes les définitions de la bibliothèque standard de Coq v8.3 en n'utilisant que trois niveaux d'univers. Il est sans doute possible d'encoder la grande majorité des preuves de Coq en fixant un nombre très bas de niveaux d'univers et en assignant des niveaux fixes à tous les termes. C'est par exemple le choix du projet Matita. Mais cette solution nécessitera sans doute la duplication de termes à des niveaux différents, dû au polymorphisme d'univers. Considérons par exemple le type des listes de Coq :

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.
```

Le type `list nat` est de type `Set`, alors que le type `list True` est de type `Prop`. Pour tout i, `list A` est dans un univers dont le niveau est i si A est dans un univers de niveau i. Le polymorphisme d'univers permet ainsi de dédoubler les types inductifs à la demande.

Une autre solution possible consiste à encoder les contraintes d'univers sous forme d'un problème de typage sous un ensemble de règles de réécriture approprié.

Un autre axe d'investigations consiste à transposer les technologies de Dedukti à d'autres contextes ou appliquer Dedukti à d'autres contextes.

Grâce au typage hors contexte des termes, Dedukti peut déjà être utilisé interactivement de la même manière que les descendants de LCF et de HOL, dans une session interactive `ghci` du compilateur GHC. Pour un usage interactif, il devient intéressant de fournir à l'utilisateur des tactiques lui permettant de construire les termes de preuve de manière semi-automatique, et en particulier de lui permettre de construire des preuves en laissant des « trous », auxquels il reviendra plus tard selon sa convenance. Il serait intéressant de chercher à transposer l'approche qui consiste à tout représenter en HOAS à des langages de termes plus complexes, en particulier les termes incluant des métavariables pour représenter des trous dans le terme de preuve partiel présenté à l'utilisateur dans un usage interactif.

Moteurs de réécriture Nous remarquons que l'extensibilité de la relation de conversion du $\lambda\Pi$-modulo offre une souplesse d'utilisation enviable pour l'utilisateur. En effet, un lemme d'égalité tel que

$$\mathsf{plus0} : \Pi n : \mathsf{nat}.\ \mathsf{plus}\ n\ 0 = n$$

peut être orienté pour donner une règle de réécriture avec laquelle on peut étendre la conversion. Ainsi, ce lemme d'égalité sera appliqué automatiquement par Dedukti partout où cela est possible, affranchissant l'utilisateur d'avoir à invoquer le lemme plus0 explicitement dans ses preuves.

Nous pourrions donc concevoir d'utiliser Dedukti non plus comme un outil très en aval de l'utilisateur mais comme un outil dans lequel il est possible de développer des preuves directement. Mais pour être efficace, ce type d'usage demande de développer un support spécifique pour certains types de règles de réécriture, tel que les règles associatives et associatives-commutatives. Comme noté dans le chapitre 3, il serait tout à fait possible de réutiliser un environnement de réécriture existant tels que ASF+SDF (van den Brand et al., 2001), Maude (Clavel et al., 2003) ou Tom (Balland et al., 2007), au prix toutefois d'avoir à encoder les termes d'ordre supérieur dans ces systèmes du premier ordre. Nous sommes particulièrement intéressé par les possibilité de combiner Dedukti avec Moca, qui offre un support spécifique des règles associatives-commutatives dans OCaml.

Entiers et tableaux primitifs Nous développons actuellement une branche expérimentale de Dedukti intégrant un support d'entiers natifs et de tableaux natifs. Il s'agit de nouvelles primitives permettant de représenter efficacement de grosses structures et calculer rapidement avec des entiers, en particulier pour vérifier des traces de solveurs SAT (Armand et al., 2010

et Jouannaud et al., 2010). Ces primitives ne sont qu'une instance particulière de l'utilisation d'une interface de fonctions étrangères (FFI) dans Dedukti, sur le modèle des FFI de Haskell ou de OCamlet d'autres langages de programmation. La partie modulo du $\lambda\Pi$-calcul modulo offre une manière particulièrement naturelle d'intégrer ces nouvelles primitives au calcul.

Bibliographie

Abadi, M., Cardelli, L., Curien, P. L. et Lévy, J. J. (1991). Explicit substitutions. *Journal of functional programming, 1*(04), 375–416.

Abel, A. (2008). Weak $\beta\eta$-normalization and normalization by evaluation for System F. Dans Cervesato, I., Veith, H. et Voronkov, A., éditeurs, *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2008*, numéro 5330 de Lecture Notes in Artificial Intelligence, pages 497–511.

Abel, A. (2009). Type structures and normalization by evaluation for System Fω. Dans Grädel, E. et Kahle, R., éditeurs, *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Proceedings*, numéro 5771 de Lecture Notes in Computer Science, pages 40–54. Coimbra, Portugal.

Abel, A. (2010). Towards normalization by evaluation for the $\beta\eta$-Calculus of Constructions. Dans (Blume et al., 2010), pages 224-239.

Abel, A., Aehlig, K. et Dybjer, P. (2007a). Normalization by evaluation for Martin-Löf type theory with one universe. Dans Fiore, M., éditeur, *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIII)*, numéro 173 de Electronic Notes in Theoretical Computer Science, pages 17–39. New Orleans, LA.

Abel, A., Coquand, T. et Dybjer, P. (2007b). Normalization by evaluation for Martin-Löf Type Theory with typed equality judgements. Dans *Proceedings of the 22nd IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 3–12.

Abel, A. et Pientka, B. (2010). Explicit substitutions for contextual type theory. Dans (Crary et Miculan, 2010).

Aehlig, K., Haftmann, F. et Nipkow, T. (2008). A compiled implementation of normalization by evaluation. Dans Mohamed, O., Muñoz, C. et Tahar, S., éditeurs, *Theorem Proving in Higher Order Logics*, numéro 5170 de Lecture Notes in Computer Science, pages 39-54. Springer Berlin / Heidelberg.

Aehlig, K. et Joachimski, F. (2004). Operational aspects of untyped Normalisation by Evaluation. *Mathematical Structures in Computer Science, 14*(04), 587–611.

Aho, A., Sethi, R. et Ullman, J. (1986). *Compilers: principles, techniques, and tools*. Reading, MA.

Allen, S. F., Bickford, M., Constable, R. L., Eaton, R. et Kreitz, C. (2003). A Nuprl-PVS connection: Integrating libraries of formal mathematics. Rapport technique Cornell University.

Altenkirch, T., Danielsson, N., Löh, A. et Oury, N. (2010). $\Pi\Sigma$: Dependent Types without the Sugar. *Functional and Logic Programming*, pp. 40–55.

Altenkirch, T., Dybjer, P., Hofmann, M. et Scott, P. (2001). Normalization by evaluation for typed lambda calculus with coproducts. Dans *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 203–210.

Altenkirch, T., Hofmann, M. et Streicher, T. (1995). Categorical reconstruction of a reduction free normalization proof. Dans *Category Theory and Computer Science*, pages 182–199. Springer-Verlag.

Altenkirch, T., Hofmann, M. et Streicher, T. (2002). Reduction-free normalisation for a polymorphic system. Dans *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science, LICS'96*, pages 98–106. IEEE.

Altenkirch, T. et McBride, C., éditeurs (2007). *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, numéro 4502 de Lecture Notes in Computer Science .

Appel, A. et MacQueen, D. (1991). Standard ML of new jersey. Dans *Programming Language Implementation and Logic Programming*, pages 1–13. Springer-Verlag.

Appel, A. W. (1998). SSA is functional programming. *ACM SIGPLAN Notices, 33*(4), 17–20.

Armand, M., Grégoire, B., Spiwack, A. et Théry, L. (2010). Extending coq with imperative features and its application to sat verication. Dans *Interactive Theorem Proving, international Conference, ITP 2010, Edinburgh, Scotland, July 11-14, 2010, Proceedings* Lecture Notes in Computer Science.

Augustsson, L. (1985). Compiling pattern matching. Dans Jouannaud, J.-P., éditeur, *Functional Programming Languages and Computer Architecture*, numéro 201 de Lecture Notes in Computer Science, pages 368-381. Springer-Verlag.

Avron, A., Honsell, F. et Mason, I. A. (1989). An overview of the Edinburgh logical framework. Dans *Current trends in hardware verification and automated theorem proving*, pages 340. Springer-Verlag.

Aydemir, B., Charguéraud, A., Pierce, B. C., Pollack, R. et Weirich, S. (2008). Engineering formal metatheory. Dans *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 3–15. ACM.

Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N. et Pierce, B. C. et al. (2005). Mechanized metatheory for the masses: The POPLMARK challenge. *Theorem Proving in Higher Order Logics*, pp. 50–65.

Balat, V., Cosmo, R. D. et Fiore, M. P. (2004). Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. Dans (Jones et Leroy, 2004), pages 64-76.

Balland, E., Brauner, P., Kopetz, R., Moreau, P. E. et Reilles, A. (2007). Tom: Piggybacking rewriting on java. *Term Rewriting and Applications*, pp. 36–47.

Barendregt, H. (1991). Introduction to generalized type systems. *Journal of functional programming, 1*(2), 125–154.

Barendregt, H. et Nipkow, T., éditeurs (1994). *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, numéro 806 de Lecture Notes in Computer Science .

Barendregt, H. P. (1985). *The Lambda Calculus, its Syntax and Semantics* Numéro 103 in Studies in Logic and the Foundations of Mathematics. Revised edition North Holland.

Barendregt, H. P. (1992). Lambda calculi with types. *Handbook of logic in computer science, 2*, 117–309.

Barendregt, H. P. (2004). The Lambda Calculus, Its Syntax and Semantics — addenda for the sixth imprinting.

Barral, F. (2008). *Decidability for Non-Standard Conversions in Typed Lambda-Calculi*. Thèse de doctorat, Université de Toulouse III - Paul Sabatier et Ludwig-Maximilian-Universität München.

Barras, B. (1996). Pure type systems. Contribution utilisateur à Coq.

Barras, B. (2000). Programming and computing in HOL. *Theorem Proving in Higher Order Logics*, pp. 17–37.

Barras, B. et Grégoire, B. (2005). On the role of type decorations in the calculus of inductive constructions. Dans *Computer Science Logic*, pages 151–166. Springer Berlin / Heidelberg.

Barras, B. et Werner, B. (1997). Coq in coq. Rapport technique INRIA.

Barthe, G., Hatcliff, J. et Sørensen, M. H. B. (1999). CPS translations and applications: the cube and beyond. *Higher-Order and Symbolic Computation, 12*(2), 125–170.

Barthe, G. et Sørensen, M. H. (2000). Domain-free pure type systems. *Journal of functional programming, 10*(05), 417–452.

Berger, U., Berghofer, S., Letouzey, P. et Schwichtenberg, H. (2006). Program extraction from normalization proofs. *Studia Logica, 82*(1), 25–49.

Berger, U., Eberl, M. et Schwichtenberg, H. (1998). Normalization by evaluation. *Prospects for Hardware Foundations*, pp. 117–137.

Berger, U., Eberl, M. et Schwichtenberg, H. (2003). Term rewriting for normalization by evaluation. *Information and Computation, 183*(1), 19–42.

Berghofer, S., Nipkow, T., Urban, C. et Wenzel, M., éditeurs (2009). *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Proceedings*, numéro 5674 de Lecture Notes in Computer Science . Munich, Germany.

Besson, F. (2006). Fast reflexive arithmetic tactics the linear case and beyond. Dans (Altenkirch et McBride, 2007), pages 48-62.

Biernacka, M. et Danvy, O. (2007). A concrete framework for environment machines. *ACM Trans. Comput. Log., 9*(1).

Biernacka, M. et Danvy, O. (2009). Towards compatible and interderivable semantic specifications for the scheme programming language, part ii: Reduction semantics and abstract machines. Dans (Palsberg, 2009), pages 186-206.

Blanqui, F. (2005). Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science, 15*(01), 37–92.

Blanqui, F., Coupet-Grimal, S., Delobel, W., Hinderer, S. et Koprowski, A. (2006). CoLoR, a Coq library on rewriting and termination. Dans *Workshop on Termination*.

Blanqui, F., Hardin, T. et Weis, P. (2007). On the implementation of construction functions for non-free concrete data types. Dans De Nicola, R., éditeur, *Programming Languages and Systems*, numéro 4421 de Lecture Notes in Computer Science, pages 95-109. Springer Berlin / Heidelberg.

Blume, M., Kobayashi, N. et Vidal, G., éditeurs (2010). *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Proceedings*, numéro 6009 de Lecture Notes in Computer Science . Sendai, Japan.

Boespflug, M. (2009). From self-interpreters to normalization by evaluation. Dans (Danvy, 2009a), pages 35–38. http://www.brics.dk/~danvy/NBE09/.

Boespflug, M. (2010). Implémentation de la machine de krivine en c. Disponible à l'adresse http://www.lix.polytechnique.fr/~mboes/projects/ckrivine/main.c.

Böhm, C. et Berarducci, A. (1985). Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science, 39*, 135–154.

Bondorf, A. et Danvy, O. (1991). Automatic autoprojection of recursive equations with global variable and abstract data types. *Science of Computer Programming, 16*(2), 151–195.

Boole, G. (1854). *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*. Macmillan.

Bove, A., Dybjer, P. et Norell, U. (2009). A brief overview of agda - a functional language with dependent types. Dans (Berghofer et al., 2009), pages 73-78.

Braibant, T. et Pous, D. (2010). An efficient coq tactic for deciding Kleene algebras. *Interactive Theorem Proving*, pp. 163–178.

Burris, S. et Lee, S. (1993). Tarski's high school identities. *American Mathematical Monthly*, pp. 231–236.

Chakravarty, M. M. T., Keller, G. et Zadarnowski, P. (2004). A functional perspective on SSA optimisation algorithms. *Electronic Notes in Theoretical Computer Science, 82*(2), 347–361.

Chapman, J., Altenkirch, T. et McBride, C. (2005). Epigram reloaded: a standalone typechecker for ett. Dans (van Eekelen, 2005), pages 79-94.

Chlipala, A. (2008). Parametric higher-order abstract syntax for mechanized semantics. Dans *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 143–156. ACM.

Chlipala, A. (2010a). Certified programming with dependent types. Projet de livre, disponible à l'adresse http://adam.chlipala.net/cpdt.

Chlipala, A. (2010b). A verified compiler for an impure functional language. Dans *POPL'10: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Chlipala, A., Malecha, G., Morrisett, G., Shinnar, A. et Wisnesky, R. (2009). Effective interactive proofs for higher-order imperative programs. *ACM SIGPLAN Notices, 44*(9), 79–90.

Church, A. (1932). A set of postulates for the foundation of logic. *Annals of Mathematics, 2*, 33-346.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics, 58*(2), 345-363.

Church, A. (1940). A formulation of the simple theory of types. *The Journal of Symbolic Logic, 5*(2), 56–68.

Church, A. (1941). *The Calculi of Lambda Conversion. Annals of Mathematics Studies*. Princeton, NJ, USA: Princeton University Press.

Church, A. et Rosser, J. B. (1936). Some properties of conversion. *Transactions of the American Mathematical Society, 39*(3), 472-482.

Clavel, M., Durán, F., Eker, S., Lincoln, P. et Martí-Oliet, N. et al. (2003). The maude 2.0 system. Dans *Rewriting Techniques and Applications*, pages 76–87. Springer Berlin / Heidelberg.

Contejean, E., Marché, C., Monate, B. et Urbain, X. (2003). Proving termination of rewriting with CiME. Dans *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, pages 71–73.

Contejean, É., Paskevich, A., Urbain, X., Courtieu, P. et Pons, O. et al. (2010). A3PAT, an approach for certified automated termination proofs. Dans *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 63–72. ACM.

Coquand, C. (1994). From semantics to rules: A machine assisted analysis. Dans Börger, E., Gurevich, Y. et Meinke, K., éditeurs, *Computer Science Logic*, numéro 832 de Lecture Notes in Computer Science, pages 91–105. Springer Berlin / Heidelberg.

Coquand, T. (1986). An analysis of Girard's paradox. Dans *Symposium on Logic in Computer Science*, pages 227. IEEE.

Coquand, T. (1991). An algorithm for testing conversion in type theory. Dans Huet, G. et Plotkin, G., éditeurs, *Logical frameworks*, pages 255–279. Cambridge University Press.

Coquand, T. (1996). An algorithm for type-checking dependent types. *Science of Computer Programming, 26*(1-3), 167-177.

Coquand, T. et Dybjer, P. (1997). Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science, 7*(01), 75–94.

Coquand, T. et Huet, G. (1988). The Calculus of Constructions. *Information and Computation, 76*(2-3), 95–120.

Coquand, T., Kinoshita, Y., Nordström, B. et Takeyama, M. (2009). A simple type-theoretic language: Mini-tt. Dans Yves Bertot Gérard Huet, J.-J. L. et Plotkin, G., éditeurs, *From Semantics to Computer Science*. Cambridge University Press.

Coquand, T. et Paulin, C. (1990). Inductively defined types. Dans *Proceedings of the international conference on Computer logic*, pages 50–66. Springer-Verlag.

Coquand, T. e. a. (2010). Formath: Formalisation of Mathematics. http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ForMath.

Cousineau, D. et Dowek, G. (2007). Embedding pure type systems in the lambda-Pi-calculus modulo. Dans *Proceedings of the 8th international conference on Typed lambda calculi and applications*, pages 102–117.

Crary, K. et Miculan, M., éditeurs (2010). *5th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice* .

Crégut, P. (1990). An abstract machine for lambda-terms normalization. Dans *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 333–340. ACM.

Crégut, P. (2004). Une procédure de décision réflexive pour un fragment de l'arithmétique de Presburger. Dans *Journées Francophones des Langages Applicatifs*.

Crégut, P. (2007). Strongly reducing variants of the krivine abstract machine. *Higher-Order and Symbolic Computation, 20*(3), 209–230.

Curien, P. L. (1991). An abstract framework for environment machines. *Theoretical Computer Science, 82*(2), 389–402.

Curien, P. L. (1993). *Categorical combinators, sequential algorithms, and functional programming*. Birkhauser.

Curry, H. B. (1934). Functionality in combinatory logic. *Proceedings of the National Academy of Science of the USA, 20*, 584-590.

Curry, H. B. et Feys, R. (1958). *Combinatory Logic*, volume 1. Amsterdam: North-Holland.

Danvy, O. (1991). Three steps for the CPS transformation. Rapport technique Manhattan, Kansas: Kansas State University.

Danvy, O. (1996). Pragmatics of type-directed partial evaluation. Dans *Selected Papers from the International Seminar on Partial Evaluation*, pages 73–94. London, UK.

Danvy, O. (1996). Type-directed partial evaluation. Dans *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* POPL '96, pages 242–257. St. Petersburg Beach, Florida, United States.

Danvy, O. (2003). A journey from interpreters to compilers and virtual machines. Dans (Pfenning et Smaragdakis, 2003), pages 117.

Danvy, O. (2008). Defunctionalized interpreters for programming languages. Dans (Hook et Thiemann, 2008), pages 131-142.

Danvy, O. (2009). From reduction-based to reduction-free normalization. *Advanced Functional Programming*, pp. 66–164.

Danvy, O., éditeur (2009a). *Informal proceedings of the 2009 Workshop on Normalization by Evaluation* . Los Angeles, CA. http://www.brics.dk/~danvy/NBE09/.

Danvy, O. (2009b). Towards compatible and interderivable semantic specifications for the scheme programming language, part i: Denotational semantics, natural semantics, and abstract machines. Dans (Palsberg, 2009), pages 162-185.

Danvy, O. et Filinski, A. (1990). Abstracting control. Dans *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM.

Danvy, O. et Millikin, K. (2008). A rational deconstruction of Landin's SECD machine with the J operator. *CoRR, abs/0811.3231*.

Danvy, O. et Nielsen, L. R. (2003). A first-order one-pass cps transformation. *Theor. Comput. Sci., 308*(1-3), 239–257.

Danvy, O. et Nielsen, L. R. (2004). Refocusing in reduction semantics. *Electronic Notes in Theoretical Computer Science, 59*(4).

Danvy, O. et Vestergaard, R. (1996). Semantics-based compiling: A case study in type-directed partial evaluation. Dans *Programming Languages: Implementations, Logics, and Programs*, pages 182–197. Springer-Verlag.

Dargaye, Z. (2007). Décurryfication certifiée. Dans *Journées Francophones des Langages Applicatifs (JFLA'07)*, pages 119–134.

Dargaye, Z. et Leroy, X. (2009). A verified framework for higher-order uncurrying optimizations. *Higher-Order and Symbolic Computation, 22*(3), 199–231.

De Bruijn, N. (1970). The mathematical language automath, its usage, and some of its extensions. Dans Laudet, M., Lacombe, D., Nolin, L. et Schützenberger, M., éditeurs, *Symposium on Automatic Demonstration*, numéro 125 de Lecture Notes in Mathematics, pages 29-61. Springer-Verlag.

De Bruijn, N. (1991). A plea for weaker frameworks. Dans *Logical frameworks*, pages 40–67.

De Bruijn, N. G. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings), 75*(5), 381-392.

De Bruijn, N. G. (1978). A namefree lambda calculus with facilities for internal definitions of expressions and segments. Rapport technique Technological University Eindhoven, Netherlands, Department of Mathematics.

De Bruijn, N. G. (1980). A survey of the project AUTOMATH. Dans Seldin, J. P. et Hindley, J. R., éditeurs, *To H.B. Curry: essays on combinatory logic, lambda calculus and formalism*, pages 589–606. Academic Press.

Delahaye, D. (2000). A tactic language for the system Coq. Dans *Logic for Programming and Automated Reasoning*, pages 377–440. Springer Berlin / Heidelberg.

Delahaye, D. (2002). A Proof Dedicated Meta-Language. *Electronic Notes in Theoretical Computer Science, 70*(2), 96–109.

Dénès, M. (2010). Compilation native de la réduction forte du Calcul des Constructions Inductives. Thèse de master (DEA, DESS, master), Université Paris VII.

Despeyroux, J., Felty, A. et Hirschowitz, A. (1995). Higher-order abstract syntax in Coq. *Typed Lambda Calculi and Applications*, pp. 124–138.

Deutsch, D. (1985). Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences, 400*(1818), 97–117.

Dezani-Ciancaglini, M. et Plotkin, G. D., éditeurs (1995). *Second International Conference on Typed Lambda Calculi and Applications, Proceedings*, numéro 902 de Lecture Notes in Computer Science .

Donnelly, K. et Xi, H. (2007). A formalization of strong normalization for simply-typed lambda-calculus and system F. *Electronic Notes in Theoretical Computer Science, 174*(5), 109–125.

Dowek, G. (2009). On the convergence of reduction-based and model-based methods in proof theory. *Logic Journal of the IGPL, 17*(5), 489-497.

Dowek, G., Hardin, T. et Kirchner, C. (2003). Theorem proving modulo. *Journal of Automated Reasoning, 31*(1), 33-72.

Dybjer, P., Nordström, B. et Smith, J. M., éditeurs (1995). *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, numéro 996 de Lecture Notes in Computer Science .

Ehrig, H., Floyd, C., Nivat, M. et Thatcher, J. W., éditeurs (1985). *Mathematical Foundations of Software Development, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin, Germany, March 25-29, 1985, Volume 1: Colloquium on Trees in Algebra and Programming (CAAP'85)*, numéro 185 de Lecture Notes in Computer Science .

Équipe de développement Coq. The coq proof assistant. http://coq.inria.fr.

Felleisen, M. (1987). *The calculi of lambda-nu-cs conversion: a syntactic theory of control and state in imperative higher-order programming languages*. Thèse de doctorat, Indianapolis, IN, USA.

Filinski, A. et Rohde, H. K. (2004). A denotational account of untyped normalization by evaluation. Dans *Proceedings of the 7th International Conference in Foundations of Software Science and Computation Structures*.

Fiore, M., Di Cosmo, R. et Balat, V. (2006). Remarks on isomorphisms in typed lambda calculi with empty and sum types. *Annals of Pure and Applied Logic, 141*(1-2), 35–50.

Fiore, M. et Simpson, A. (1999). Lambda definability with sums via Grothendieck logical relations. *Typed Lambda Calculi and Applications*, pp. 644–644.

Flanagan, C., Sabry, A., Duba, B. F. et Felleisen, M. (1993). The essence of compiling with continuations (with retrospective). Dans (McKinley, 2004), pages 502-514.

Fluet, M. et Weeks, S. (2001). Contification using dominators. *ACM SIGPLAN Notices, 36*(10), 2–13.

Frege, G. (1879). *Begriffsschrift: eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.*. Halle.

Gabbay, D. M. et Olivetti, N. (2000). Goal-directed proof theory. *Kluwer Applied Logic Series; Vol. 21*, pp. 266.

Garillot, F. et Werner, B. (2007). Simple types in type theory: Deep and shallow encodings. Dans Schneider, K. et Brandt, J., éditeurs, *Theorem Proving in Higher Order Logics*, numéro 4732 de Lecture Notes in Computer Science, pages 368-382. Springer Berlin / Heidelberg.

Garrigue, J. (1998). Programming with polymorphic variants. Dans *ML Workshop*.

Gentzen, G. (1934). Untersuchungen über das logische schliessen. *Mathematische Zeitschrift, 39*, 176-210. Translated in Szabo, editor., The Collected Papers of Gerhard Gentzen as « Investigations into Logical Deduction ».

Geuvers, H. (1994). A short and flexible proof of strong normalization for the calculus of constructions. Dans (Dybjer et al., 1995), pages 14-38.

Geuvers, H., Krebbers, R., McKinna, J. et Wiedijk, F. (2010). Pure type systems without explicit contexts. *CoRR, abs/1009.2792*.

Geuvers, H., McKinna, J. et Wiedijk, F. (2009). Pure type systems without explicit contexts. Submitted.

Ghani, N. (1995). $\beta\eta$-equality for coproducts. *Typed Lambda Calculi and Applications*, pp. 171–185.

Gill, A. (2009). Type-safe observable sharing in Haskell. Dans *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 117–128. ACM.

Girard, J. Y. (1971). Une extension de l'interpretation de Gödel à l'analyse, et son application à l'Élimination des coupures dans l'analyse et la théorie des types. Dans *Proceedings of the Second Scandinavian Logic Symposium*, pages 63-–92.

Gödel, K. (1931). Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik, 38*(1), 173-198.

Gödel, K. (1958). Uber eine bisher noch nicht benutzte Erweiterung des finiten Standpunktes. *Dialectica, 12*(280-287), 12.

Gonthier, G. (2007). The four colour theorem: Engineering of a formal proof. Dans (Kapur, 2007), pages 333.

Gonthier, G. et Mahboubi, A. (2008). A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA.

Gordon, M. (2000). From LCF to HOL: a short history. Dans Plotkin, G., Stirling, C. et Tofte, M., éditeurs, *Proof, language, and interaction: essays in honour of Robin Milner*, pages 169–185. The MIT Press.

Gordon, M. et Melham, T. (1993). *Introduction to HOL: A theorem proving environment for higher order logic*. New York, NY, USA: Cambridge University Press.

Gordon, M. J. C., Milner, R. et Wadsworth, C. P. (1979). *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 de *Lecture Notes in Computer Science*. Springer-Verlag.

Grégoire, B. (2003). *Compilation des termes de preuves: un (nouveau) mariage entre Coq et OCaml*. Thèse de doctorat, Université Denis Diderot – Paris 7.

Grégoire, B. et Leroy, X. (2002). A compiled implementation of strong reduction. Dans *Proceedings of the seventh ACM SIGPLAN international conference on functional programming*, pages 235–246.

Grégoire, B. et Mahboubi, A. (2005). Proving equalities in a commutative ring done right in coq. Dans Hurd, J. et Melham, T., éditeurs, *Theorem Proving in Higher Order Logics*, numéro 3603 de Lecture Notes in Computer Science, pages 98-113. Springer Berlin / Heidelberg.

Gunter, C. A., Rémy, D. et Riecke, J. G. (1995). A generalization of exceptions and control in ML-like languages. Dans *Proceedings of the seventh international conference on functional programming languages and computer architecture*, pages 12–23. ACM.

Gurevič, R. (1985). Equational theory of positive numbers with exponentiation. *Proceedings of the American Mathematical Society, 94*(1), 135–141.

Hales, T. C. (2005). Introduction to the flyspeck project. Dans *Mathematics, Algorithms, Proofs*.

Hardin, T. et Lévy, J. J. (1989). A confluent calculus of substitutions. Dans *France-Japan Artificial Intelligence and Computer Science Symposium*.

Harper, R. et Licata, D. R. (2007). Mechanizing metatheory in a logical framework. *Journal of Functional Programming, 17*(4-5), 613–673.

Hasenjaeger, G. et Scholz, H. (1961). *Grundzüge der mathematischen Logik*. Springer-Verlag.

Hatcliff, J. et Danvy, O. (1994). A generic account of continuation-passing styles. Dans *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 458-471.

Hatcliff, J. et Danvy, O. (1997). A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, pp. 507-541.

Helmink, L. et Ahn, R. (1991). Goal directed proof construction in type theory. Dans *Logical frameworks*, pages 120–148. Cambridge University Press.

Hilbert, D. (1927). *The foundation of mathematics*..

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM, 12*, 576–580.

Hook, J. et Thiemann, P., éditeurs (2008). *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008* . Victoria, BC.

Howard, W. (1980). The formulas-as-types notion of construction. *To HB Curry: Essays on combinatory logic, lambda-calculus and formalism*, pp. 479–490.

Howe, D. (1996). Importing mathematics from HOL into Nuprl. *Theorem Proving in Higher Order Logics*, pp. 267–281.

Huet, G. (1989). The constructive engine. Dans Narasimhan, R., éditeur, *A Perspective in theoretical computer science: commemorative volume for Gift Siromoney* World scientific series in computer science. World Scientific.

Huet, G. (1997). The zipper. *Journal of Functional Programming, 7*(05), 549–554.

Johnsson, T. (1985). Lambda lifting: transforming programs to recursive equations. Dans *Functional Programming Languages and Computer Architecture*, pages 190–203.

Jones, N. D. et Leroy, X., éditeurs (2004). *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004* . Venice, Italy.

Joshi, S. et Mahboubi, A. (2009). Cooper's quantifier elimination for presburger arithmetic in Coq. Non publié.

Jouannaud, J.-P., Strub, P.-Y. et Zhang, L. (2010). Certification of SAT solvers in Coq. Rapport technique INRIA.

Kapur, D., éditeur (2007). *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Revised and Invited Papers*, numéro 5081 de Lecture Notes in Computer Science . Singapore.

Keller, C. (2009). Importation de preuves HOL-Light en Coq. Thèse de master (DEA, DESS, master), Université Paris VII.

Keller, C. et Werner, B. (2010). Importing HOL Light into Coq. Dans , pages 307–322.

Kelsey, R. A. (1995). A correspondence between continuation passing style and static single assignment form. Dans *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, pages 22. ACM.

Kfoury, A. et Wells, J. (1994). A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. *ACM SIGPLAN Lisp Pointers, 7*(3), 207.

Kirchner, C. et Kirchner, H. (1999). Rewriting, solving, proving. A preliminary version of a book available at http://www.loria.fr/~ckirchne/=rsp/rsp.pdf.

Kleene, S. C. et Rosser, J. B. (1936). The inconsistency of certain formal logics. *Annals of Math., 2*(36), 630-636.

Klop, J. W. (1980). *Combinatory reduction systems*. Thèse de doctorat, CWI.

Krivine, J. L. (1985). Un interpréteur du lambda-calcul. Brouillon. Disponible à l'adresse http://www.pps.jussieu.fr/~krivine/articles/interprt.pdf.

Krivine, J. L. (2007). A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation, 20*(3), 199–207.

Lafont, Y. (1988). *Logique, catégories et machines : implantation de langages de programmation guidée par la logique catégorique*. Thèse de doctorat, Université Paris VII.

Landin, P. J. (1964). The mechanical evaluation of expressions. *The Computer Journal, 6*(4), 308.

Le Fessant, F. et Maranget, L. (2001). Optimizing pattern matching. Dans *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 26–37.

Leroy, X. (1990). The ZINC experiment: an economical implementation of the ML language. Rapport technique INRIA.

Leroy, X. (2005). From Krivine's machine to the Caml implementations. Invited talk given at the KAZAM workshop.

Leroy, X. (2007). Formal verification of an optimizing compiler. Dans Baader, F., éditeur, *Term Rewriting and Applications*, numéro 4533 de Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

Leroy, X., Doligez, D., Garrique, J., Rémy, D. et Vouillon, J. (2009). Objective caml, release 3.11.1. http://caml.inria.fr/ocaml/.

Lescuyer, S. et Conchon, S. (2008). A reflexive formalization of a SAT solver in Coq. Dans *Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*.

Lévy, J. J. (1978). *Réductions correctes et optimales dans le Lambda-Calcul*. Université Denis Diderot – Paris 7.

Lindley, S. (2005). *Normalisation by evaluation in the compilation of typed functional programming languages*. Thèse de doctorat,.

Luo, Z. (1989). Ecc, an extended calculus of constructions. Dans (Parikh, 1989), pages 386-395.

Luo, Z. et Pollack, R. (1992). *LEGO proof development system: User's manual*. LFCS, Dept. of Computer Science, University of Edinburgh.

Maranget, L. (2008). Compiling pattern matching to good decision trees. Dans *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46.

Marlow, S. et Peyton-Jones, S. (2006). Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming, 16*(4-5), 415–449.

Martin-Löf, P. (1975b). About models for intuitionistic type theories and the notion of definitional equality. Dans *Proceedings of the 3rd Scandinavian Logic Symposium*, pages 81–109.

Martin-Löf, P. (1975a). An intuitionistic theory of types: Predicative part. *Studies in Logic and the Foundations of Mathematics, 80*, 73–118.

Martin-Löf, P. (1984). *Intuitionistic type theory*. Bibliopolis.

McBride, C. (2001). The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript.

McBride, C. et McKinna, J. (2004). Functional pearl: i am not a number–i am a free variable. Dans *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 1–9. ACM.

McCullagh, D. (2010). Buggy McAfee update whacks Windows XP PCs. http://news.cnet.com/8301-1009_3-20003074-83.html.

McKinley, K. S., éditeur (2004). *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection* .

McKinna, J. et Pollack, R. (1993). Pure type systems formalized. *Typed Lambda Calculi and Applications*, pp. 289–305.

Miculan, M. (2001). Developing (Meta) Theory of lambda-calculus in the Theory of Contexts. *Electronic Notes in Theoretical Computer Science, 58*(1), 37–58.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences, 17*(3), 348–375.

Mogensen, T. A. (1992). Efficient self-interpretation in lambda calculus. *Journal of Functional Programming, 2*(03), 345–364.

Moggi, E. (1989). Computational lambda-calculus and monads. Dans (Parikh, 1989), pages 14-23.

Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P. et Birkedal, L. (2008a). Ynot: Reasoning with the awkward squad. Dans *ACM SIGPLAN International Conference on Functional Programming*.

Nanevski, A., Pfenning, F. et Pientka, B. (2008b). Contextual modal type theory. *ACM Trans. Comput. Logic, 9*, 1–49.

Naumov, P., Stehr, M. O. et Meseguer, J. The HOL/NuPRL proof translator — A practical approach to formal interoperability. Dans *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs' 2001, Edinburgh, Scotland, UK, September 3–6, 2001, Proceedings*, pages 329–345.

Newman, M. H. A. (1942). On theories with a combinatorial definition of equivalence. *Annals of Mathematics, 43*(2), 223–243.

Obua, S. et Skalberg, S. (2006). Importing HOL into ISABELLE/HOL. *Automated Reasoning*, pp. 298–302.

Palsberg, J., éditeur (2009). *Semantics and Algebraic Specification, Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday*, numéro 5700 de Lecture Notes in Computer Science .

Parikh, R., éditeur (1989). *Proceedings, Fourth Annual Symposium on Logic in Computer Science* .

Partain, W. (1992). The nofib benchmark suite of Haskell programs. Dans *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202.

Paulson, L. C. (1992). Designing a theorem prover. Dans Abramsky, S., Gabbay, D. M. et Maibaum, T. S. E., éditeurs, *Handbook of Logic in Computer Science*, pages 415-475. Oxford University Press.

Peano, G. (1889). *The principles of arithmetic, presented by a new method.*.

Peyton-Jones, S. (1987). *The Implementation of Functional Programming Languages*. Prentice/Hall International.

Peyton-Jones, S. (1992). Implementing lazy functional languages on stock hardware: the spineless tagless G-machine version 2.5. *Journal of Functional Programming, 2*(2), 127–202.

Peyton-Jones, S. (2003). Special issue: Haskell 98 language and libraries. *Journal of Functional Programming, 13*(1).

Peyton Jones, S., Partain, W. et Santos, A. (1996). Let-floating: moving bindings to give faster programs. Dans *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 1–12. ACM.

Pfenning, F. et Elliott, C. (1988). Higher-order abstract syntax. Dans *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation - PLDI '88*, pages 199–208. Atlanta, Georgia, United States.

Pfenning, F. et Schürmann, C. (1999). System description: Twelf—a meta-logical framework for deductive systems. *Automated Deduction—CADE-16*, pp. 679–679.

Pfenning, F. et Smaragdakis, Y., éditeurs (2003). *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Proceedings*, numéro 2830 de Lecture Notes in Computer Science . Erfurt, Germany.

Pientka, B. et Dunfield, J. (2010). Beluga: A framework for programming and reasoning with deductive systems (system description). *Automated Reasoning*, pp. 15–21.

Poirier, H. (2002). La vraie nature de l'intelligence. *Sciences et Vie, 1013*.

Pollack, R. (1988). The theory of LEGO. Manuscript.

Pollack, R. (1994). Closure under alpha-conversion. Dans (Barendregt et Nipkow, 1994), pages 313–332.

Pollack, R. (1995). A verified typechecker. Dans (Dezani-Ciancaglini et Plotkin, 1995), pages 365-380.

Post, E. L. (1936). Finite combinatory processes-formulation 1. *Journal of Symbolic Logic, 1*(3), 103–105.

Prawitz, D. (1965). *Natural deduction: A proof-theoretical study*. Almqvist & Wiksell Stockholm.

Pugh, W. (1991). The Omega test: a fast and practical integer programming algorithm for dependence analysis. Dans *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 13. ACM.

Revesz, G. (1985). Axioms for the theory of lambda-conversion. *SIAM Journal on Computing, 14*, 373.

Reynolds, J. C. (1972). Definitional interpreters for higher-order programming languages. Dans *Proceedings of the ACM Annual Conference*, pages 717–740. New York, NY, USA. Republié en tant que (Reynolds, 1998)..

Reynolds, J. C. (1974). Towards a theory of type structure. Dans *Programming Symposium*, pages 408–425. Springer-Verlag.

Reynolds, J. C. (1985). Three approaches to type structure. Dans (Ehrig et al., 1985), pages 97-138.

Reynolds, J. C. (1998). Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation, 11*(4), 363–397.

Schürmann, C. et Stehr, M. O. (2006). An executable formalization of the HOL/Nuprl connection in the metalogical framework Twelf. Dans *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 150–166.

Scott, D. S. (1993). A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science, 121*(1-2), 411–440.

Selinger, P. (2003). From continuation passing style to Krivine's abstract machine. Manuscrit. Disponible à l'adresse http://www.mscs.dal.ca/~selinger/papers/krivine.pdf.

Severi, P. et Poll, E. (1994). Pure type systems with definitions. *Logical Foundations of Computer Science*, pp. 316–328.

Sheard, T. (1999). Using MetaML: A staged programming language. *Advanced Functional Programming*, pp. 207–239.

Sheard, T. et Peyton-Jones, S. (2002). Template meta-programming for Haskell. *ACM SIGPLAN Notices, 37*(12), 60–75.

Silès, V. et Herbelin, H. (2010). Pure Type System conversion is always typable. Submitted.

Steele Jr., G. L. (1978). Rabbit: A compiler for scheme. Rapport technique Cambridge, MA, USA: Massachusetts Institute of Technology.

Taha, W. (2004). A gentle introduction to multi-stage programming. Dans Lengauer, C., Batory, D., Consel, C. et Odersky, M., éditeurs, *Domain-Specific Program Generation*, numéro 3016 de Lecture Notes in Computer Science, pages 30-50. Springer Berlin / Heidelberg.

Takahashi, M. (1995). Parallel reductions in λ-calculus. *Information and Compututation, 118*, 120-127.

The Dedukti Development Team (2009). Dedukti, a universal proof checker, version 1.0. http://www.lix.polytechnique.fr/dedukti/.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc., 2*(42), 230-265.

van Benthem Jutting, L. S. (1993). Typing in pure type systems. *Inf. Comput., 105*(1), 30–41.

van Benthem Jutting, L. S., McKinna, J. et Pollack, R. (1993). Checking algorithms for pure type systems. Dans (Barendregt et Nipkow, 1994), pages 19-61.

van den Brand, M., van Deursen, A., Heering, J., de Jong, H. A. et de Jonge, M. et al. (2001). The asf+sdf meta-environment: A component-based language development environment. Dans (Wilhelm, 2001), pages 365-370.

van Eekelen, M. C. J. D., éditeur (2005). *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005* . Tallinn, Estonia.

van Heijenoort, J. (1977). *From Frege to Gödel: a source book in mathematical logic, 1879-1931*. Harvard University Press.

Vestergaard, R. (2001). The polymorphic type theory of normalisation by evaluation. Manuscript disponible à l'adresse http://www.jaist.ac.jp/~vester/Writings/nbe-f.ps.gz.

Wadler, P. (1989). Theorems for free!. Dans *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM.

Wadler, P. (1990). Deforestation: transforming programs to eliminate trees. *Theoretical computer science, 73*(2), 231–248.

Werner, B. (1994). *Une Théorie des Constructions Inductives*. Thèse de doctorat, Université Paris-Diderot - Paris VII.

Whitehead, A. N. et Russell, B. (1910). *Principia Mathematica*. Cambridge University Press.

Wiedijk, F. (2009). Stateless HOL. Dans *Liber Amicorum for Roel de Vrijer, Vrije Universiteit Amsterdam*, pages 227–240. Citeseer.

Wilhelm, R., éditeur (2001). *Compiler Construction, 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, Proceedings*, numéro 2027 de Lecture Notes in Computer Science . Genova, Italy.

Wilkie, A. (2000). On exponentiation — a solution to Tarski's high school algebra problem. Rapport technique Dept. Math., Secunda Univ. Napoli, Caserta.